QF-Test - The Tutorial

Version 4.0.8

Karlheinz Kellerer, Martin Moser, Michael Lehto, Magnus Lindström
Quality First Software GmbH[1]

Copyright © 2002-2015 Quality First Software GmbH

November 4, 2015

[1] http://www.qfs.de

Preface

This tutorial is meant as a hands-on introduction to QF-Test.

In the base parts 1 and 2 we will show the main functions of QF-Test and guide you through the necessary steps to set up your own test-suite. You will learn how to analyse your test results, step through your test by use of the debugger and generate an overview report. Further topics are the concept of modularization by help of procedures and component recognition, which is of central meaning in GUI Testing.

Dependent on whether your system under test is based on Java or Web you can decide to work through part 1 or 2 or both if needed.

Part 3 leads to some more advanced QF-Test features like data driven testing, ensuring of test-case prerequisites and automatic generation of basic procedures.

Working through the basic tutorial should take you 4.5 up to 6.5 hours, dependent on your previous knowledge. The estimated duration for each chapter is indicated at its beginning.

As this tutorial will grow along with QF-Test with new examples being added as new features become available, chapters are written to work independently of each other. So you can always come back and work through some new chapters without having to go through the previous ones.

This tutorial is also available as HTML online version at http://www.qfs.de/en/qftest/tutorial.html.

As an alternative to private study, QFS offers training courses for QF-Test. Details can be found at http://www.qfs.de/en/qftest/training.html.

Feedback

In order to make this tutorial more useful we need your help. It is always hard for the developer of a product to guess which features are important to the users; which of them are easy to understand and use and which remain a mystery. That is why we want to encourage you to give us feedback about what you would like to see in this tutorial.

Of course we would also like to hear about any problems you have, either because the examples don't work for you or because you find them missing the point or whatever. In the end it is your feedback that determines the quality of this tutorial. Please send all comments, bug reports, wishes etc. to qftest@qfs.de.

Contents

I Java testing with QF-Test 1

1 Working with a Sample Test-suite [30-45 min] 2

1.1 Before you start . . . 2
1.2 Starting QF-Test and Loading the Test-suite . . . 2
1.3 Starting the Application . . . 4
1.4 Clickstream Test-case . . . 7
1.5 A Few Tips . . . 9
1.6 Text Check . . . 9
1.7 Checking a Radio Button . . . 15
1.8 Stopping the Application . . . 16
1.9 The Complete Test . . . 18
1.10 Report Generation . . . 18

2 Creating a Test-suite [45-60 min] 21

2.1 Introduction . . . 21
2.2 Starting the Application . . . 21
2.3 Adding a Clickstream Test . . . 30
2.4 Building a Test-suite . . . 31
2.5 Adding a Text-Check . . . 34
2.6 Checking Business Logic . . . 38

3 Using the Debugger [30-45 min] 40

3.1 Introduction . . . 40
3.2 Starting the Debugger . . . 40

3.3 The Debugger Window . . . 42
3.4 The Current Node versus the Selected Node . . . 43
3.5 Stepping Through a Test or Sequence . . . 44
3.6 Skipping Execution of Nodes . . . 47
3.7 Setting Breakpoints . . . 48
3.8 Resolving Run-Time Problems . . . 49
3.9 Jump to Run Log . . . 51

4 Writing a Procedure [30-45 min] 52
4.1 Getting Started with a New Suite . . . 52
4.2 Create the Procedure . . . 55
4.3 Add a Check Node . . . 55
4.4 About Components . . . 57
4.5 The Try/Catch Mechanism . . . 58
4.6 Complete the Selection Logic . . . 61
4.7 Making the Procedure More Usable . . . 62
4.8 Calling the Procedure . . . 63

5 Creating a Generalized Procedure [30-45 min] 66
5.1 Creating a Package . . . 66
5.2 Move the Procedure . . . 67
5.3 Create the Deselect Procedure . . . 67
5.4 Call the Procedures . . . 68
5.5 Creating an All-Purpose Procedure . . . 69
5.6 Setting a Default . . . 69
5.7 The If/Else Construct . . . 70
5.8 Calling the All-Purpose Procedure . . . 72
5.9 Documenting the Procedure . . . 73
5.10 Saving your Test-suite . . . 76

6 Modularization [30-45 min] 77
6.1 Creating the Driver Suite . . . 77
6.2 What Belongs Where . . . 78

6.3 Creating a Test Sequence . . . 79
6.4 Calling a Procedure in the Utility Suite . . . 79
6.5 Adding an Include . . . 80
6.6 Modularization for Multiple SUTs . . . 81

7 The Standard Library [30-45 min] 83
7.1 The SUT For Testing . . . 83
7.2 The Standard Library . . . 85
7.3 Selected Packages and Procedures . . . 88
7.3.1 The Checkbox Package . . . 88
7.3.2 The Combobox/Combo Packages . . . 89
7.3.3 The General Packages . . . 89
7.3.4 The List Packages . . . 89
7.3.5 The Menu Package . . . 89
7.3.6 The Popup Menu Package . . . 91
7.3.7 The SWT/Sash Package . . . 92
7.3.8 The Table Package . . . 93
7.3.9 The Table/Selection Package . . . 93
7.3.10 The Tabbedpane/CTabfolder Package . . . 94
7.3.11 The Text Package . . . 94
7.3.12 The Tree Package . . . 95
7.3.13 The Cleanup Package . . . 96
7.3.14 The Swing/Filechooser Utility Package . . . 97
7.3.15 The Swing/Optionpane Utility Package . . . 100
7.3.16 The SWT/FileDialog Package . . . 103
7.3.17 The SWT/ColorDialog Package . . . 103
7.3.18 The SWT/DirectoryDialog Package . . . 103
7.3.19 The SWT/Instrument Package . . . 103
7.3.20 The AWT/Menu Package . . . 104
7.3.21 The Run-log Package . . . 104
7.3.22 The Run-log.Screenshots Package . . . 104
7.3.23 The Shellutils Package . . . 104

7.3.24 The Utils Package . . . 105
7.3.25 The Database Package . . . 105
7.3.26 The Check Package . . . 106
7.3.27 The Databinder Package . . . 106
7.3.28 The Web Package . . . 106

8 Managing Complex GUI Components [30-45 min] 108
8.1 The SUT For Testing . . . 108
8.2 One-Dimensional Complex Components . . . 110
8.3 Two-Dimensional Complex Components . . . 113
8.4 Item Nodes versus Syntax . . . 114
8.5 The Importance of Uniqueness . . . 115
8.6 Trees . . . 116

9 It's time to start your own Application 118

II Web testing with QF-Test 119

10 Working with a Sample Test-suite [30-45 min] 120
10.1 Starting QF-Test and Loading the Test-suite . . . 120
10.2 Starting the browser . . . 122
10.3 Clickstream Test-case . . . 126
10.4 A Few Tips . . . 128
10.5 Text Check . . . 128
10.6 Checking a Radio Button . . . 133
10.7 Stopping the Browser . . . 134
10.8 The Complete Test . . . 135
10.9 Report Generation . . . 136

11 Creating a Test-suite [45-60 min] 139
11.1 Introduction . . . 139
11.2 Starting the Application . . . 139
11.3 Adding a Clickstream Test . . . 151

11.4 Building a Test-suite 152
11.5 Adding a Text Check 154

12 Further reading [5 min] 159

III Advanced features of QF-Test 160

13 Introduction [5 min] 161

14 The Demo Application [5 min] 162

15 Data driven Testing: Running one Test-case with different test data [30-45 min] 164
15.1 Situation 164
15.2 Traditional way of implementing data driven testing 165
15.3 Data driver concept 166
15.4 Summary 172

16 Dependencies: Automatically ensuring correct prerequisites for each Test-case [60 min] 174
16.1 General 174
16.2 Ensuring prerequisites 176
16.3 Combining dependencies 179
16.4 Error and exception handling 184
16.4.1 Error handling 184
16.4.2 Exception handling 185
16.4.3 Summary 188
16.5 More about dependencies 188

17 Automated creation of basic procedures [60 min] 189
17.1 General 189
17.2 Automated creation of procedures 191
17.3 Configuration of the automated creation 195
17.3.1 Introduction 195

17.3.2 First example 196
17.3.3 Using the current text for checking 201
17.3.4 Creating container procedures 203
17.3.5 Using the current value of child components 206
17.3.6 More configuration capabilities 210

List of Figures

1.1 The Test-suite `Options.qft` . . . 3
1.2 The ”Test-set: Options” Node . . . 4
1.3 The Setup Node . . . 4
1.4 The Start Java SUT Client Node . . . 5
1.5 The Options Demo . . . 7
1.6 The Clickstream Test-case Node . . . 8
1.7 The ”Table” Test Sequence . . . 8
1.8 The Text Field to Check . . . 10
1.9 The Text Check Test-case . . . 10
1.10 Error in Check Text Node . . . 11
1.11 Run-Log for the Check Text Sequence . . . 12
1.12 Diagnosis of Error in the Check Text Sequence. . . . 13
1.13 Screenshot node showing the error situation . . . 14
1.14 The ”Selected test” Node . . . 15
1.15 Radio Buttons in the Options Demo . . . 16
1.16 The Cleanup Sequence . . . 17
1.17 Run-Log for the Completed Test-set ”Options” . . . 18
1.18 Report Generation Properties . . . 19
1.19 An HTML Report . . . 20

2.1 The Quickstart Wizard . . . 22
2.2 Select type of SUT . . . 23
2.3 Specification of the Java executable. . . . 24
2.4 Working directory. . . . 25

2.5 Jar file selection. 26
2.6 SWT instrumentation 27
2.7 Client name 28
2.8 Final Information 29
2.9 Generated Setup Sequence 30
2.10 The FileChooserDemo Window 30
2.11 The Recorded Clickstream 31
2.12 Start organizing the Test-suite 32
2.13 Component selection dialog 33
2.14 Setup sequence of the FileChooserDemo 33
2.15 The Organization of your Test-suite 34
2.16 Recording a Check in the FileChooser Window 35
2.17 The Text-Check Test-case Organized into the Tree Structure 36
2.18 The Run-Log of Your Test-set 36
2.19 Properties of the ”Check Text” Node 37
2.20 The Check Enabled Status Test-case 38

3.1 Starting `Options.qft` with the debugger 41
3.2 The debugger window 43
3.3 After a step-in operation 45
3.4 Preparing for step-over command 46
3.5 Preparing for the step-out command 47
3.6 Setting a breakpoint 48
3.7 Select a node to modify 49
3.8 The modified node 49
3.9 No component for QF-Test ID error 50
3.10 Jump to run-log 51

4.1 Skeleton for new Test-suite 53
4.2 Checkbox within the Miscellaneous Options of the SUT 54
4.3 Adding a Procedure 55
4.4 Recording the ”Check boolean: selected” Node 56

4.5 A 'Check boolean: selected' Node within the Procedure 57
4.6 The Checkbox omponent within the component tree 58
4.7 'Check boolean: selected' Error 59
4.8 Throw Exception on Failure 59
4.9 Selecting an Exception to Catch 60
4.10 CheckFailedException in the Run-Log 60
4.11 Procedure with Try/Catch 61
4.12 Procedure with Completed Try/Catch Mechanism 62
4.13 Variable Replacing Fixed Component 63
4.14 Properties of a Procedure Call 64

5.1 The Checkbox Package 67
5.2 Check for Unselected State 68
5.3 Checkbox Package with two Procedures 68
5.4 Default Value for Procedure Variable 70
5.5 Condition Statement for the "If" Node 71
5.6 The If/Else Construct 71
5.7 The If/Elseif/Else Construct 72
5.8 Procedure Call to setState 72
5.9 Comments for Package Documentation 74
5.10 The Package Documentation 75

6.1 Utility Suite `utils.qft` 78
6.2 Deselect Checkbox Sequence-Node 79
6.3 Sequence-Node with Recorded Click 79
6.4 Procedure Call to utils.qft 80
6.5 An Include List 80
6.6 Constant Reference to the SUT Client 81
6.7 Variable Reference to the SUT Client 81
6.8 Setting a Default Test-suite Variable 82

7.1 The Test-suite `StdLibDemo.qft` 84
7.2 The SUT for Testing the Standard Library 85

7.3 The Standard Library . . . 86
7.4 Setting the $(client) Variable . . . 87
7.5 Example Usage of the `qfs.swing.checkbox` Package . . . 88
7.6 Example Call to `qfs.swing.menu.setSubCheckItem` . . . 90
7.7 Selecting a Sub-Menu Check Item within the SUT . . . 91
7.8 Selecting a Sub-Menu Item within a Popup Menu . . . 92
7.9 Example Call to `popupmenu.setSubItem` . . . 92
7.10 Example Usage of `text.clearArea` . . . 94
7.11 Example Usage of `tree.expand` . . . 95
7.12 Tree Component Items . . . 96
7.13 Example Usage of `qfs.swing.filechooser.selectFile` . . . 98
7.14 Recorded JFileChooser Dialogs and Components . . . 98
7.15 Using the Name Resolver . . . 99
7.16 Implementation of `selectFile` with Generic Components . . . 99
7.17 Generic FileChooser Components in `qfs.qft` . . . 99
7.18 `JOptionPane` dialog . . . 100
7.19 Generic OptionPane Components in `qfs.qft` . . . 102

8.1 Start Java SUT Client Node for ItemsDemo . . . 109
8.2 The ItemsDemo Window . . . 110
8.3 Mouse Operations Performed on a JList . . . 110
8.4 Reference to a JList Item . . . 111
8.5 JList Item Node . . . 111
8.6 JList Item Properties . . . 111
8.7 JList Item with Numeric Index . . . 112
8.8 JList Item with Regular Expression Index . . . 112
8.9 The Sample Table within the SUT . . . 113
8.10 Mouse Clicks Recorded from the JTable . . . 113
8.11 Registered Items Nodes for the JTable . . . 114
8.12 Properties of a JTable Column Item Node . . . 114
8.13 Mouse Clicks using the syntax for direct access . . . 115
8.14 A Table with Duplicate Items . . . 115

8.15 Mouse Clicks Recorded from the JTree 116
8.16 Direct Numeric Reference to a JTree Node 117

10.1 The Test-suite `webdemo.qft` 121
10.2 The ”Test-set: Web demo” Node 122
10.3 The ”Setup: Start browser” 123
10.4 The ”Start brower” Node 124
10.5 The Web Demo Page in the Browser Window 126
10.6 The Clickstream Test-case Node 127
10.7 The ”Menu items” Test Sequence 127
10.8 The ”Name:” Label to Check 129
10.9 The Text Check Test-case 129
10.10 Error in String Check Node 130
10.11 Run-Log for the Text Check Sequence 130
10.12 Diagnosis of Error in the Text Check Sequence. 131
10.13 Screenshot node showing the error situation 132
10.14 The ”Selected test” Node 133
10.15 Radio Buttons in the Web Demo 134
10.16 The Cleanup Sequence ”Stop Browser” 135
10.17 Run-Log for the Completed Test-set ”Web demo” 135
10.18 Report Generation Properties 136
10.19 An HTML Report 137

11.1 The Quickstart Wizard 140
11.2 Select type of SUT 141
11.3 Specification of the URL. 142
11.4 AJAX toolkit. 143
11.5 Browser selection. 144
11.6 Internet Explorer compatibility mode. 145
11.7 Java executable for starting the Browser wrapper. 146
11.8 Browser settings 147
11.9 Browser window name 148

11.10 Client name . . . 149
11.11 Final Information . . . 150
11.12 Generated Setup Sequence . . . 151
11.13 The Recorded Clickstream . . . 152
11.14 The Organization of your Test-suite . . . 153
11.15 Recording a Text Check . . . 155
11.16 The Text-Check Test Organized into the Tree Structure . . . 156
11.17 The Run-Log of Your Test . . . 157
11.18 Properties of the ”Check Text” Node . . . 158

14.1 The main window of the JCarConfigurator . . . 163

15.1 Traditional way of data driven testing . . . 165
15.2 Traditional way with a nested Test-set . . . 165
15.3 Data table dialog . . . 167
15.4 The filled data table . . . 168
15.5 Test-set with Data driver . . . 168
15.6 Using the `$(discount)` parameter . . . 169
15.7 Full data table . . . 170
15.8 Name for run-log and report attribute . . . 171
15.9 Run-log with different names per Test-case . . . 172

16.1 First Test-set of dependencies_work.qft . . . 175
16.2 First Test-set of dependencies_work.qft . . . 176
16.3 Sample test-suite with the first Dependency . . . 176
16.4 The run-log of the execution . . . 177
16.5 The new Setup implementation . . . 178
16.6 The test-suite with a Dependency reference . . . 178
16.7 Ensuring prerequisites for Test-case ’Discount 15’ . . . 179
16.8 ’vehicles dialog opened’ Dependency . . . 182
16.9 Implementation of specified test-cases . . . 182
16.10 Run-log of nested Dependencies . . . 183
16.11 Test-suite of Error handler . . . 184

16.12 Dependency with Error handler 184
16.13 Run-log for Dependency with Error handler 185
16.14 Try-catch nodes in each Test-case 186
16.15 Test-suite with Catch 187
16.16 Run-log of a Dependency with Catch 188

17.1 Screenshot of test-suite 191
17.2 The test-suite automated_procedures_work.qft 192
17.3 The recorded procedures 193
17.4 The test-suite containing the procedures 194
17.5 The procedures for all panels 195
17.6 The current configuration 196
17.7 The own configuration file 197
17.8 The `checkText` procedure 198
17.9 The `checkText` procedure with parameters 199
17.10 Using the `<COMPID>` place holder 200
17.11 Your first automatically created procedures 201
17.12 The configuration for the current text 202
17.13 The created procedures with the current text 203
17.14 The template for container procedures 204
17.15 Usage of `@FORCHILDREN` tag 205
17.16 The created container procedures 206
17.17 Configuration of `<CCURRENTVALUE>` 207
17.18 Test-suite using `<CCURRENTVALUE>` 208
17.19 Parameters for container procedures 209
17.20 Parameters for container procedures in test-suite 210

Part I

Java testing with QF-Test

Chapter 1

Working with a Sample Test-suite [30-45 min]

1.1 Before you start

On first startup of QF-Test and/or the System Under Test (SUT) via QF-Test you might get a security warning from the Windows firewall asking whether to block Java or not. As QF-Test communicates with the SUT by means of network protocols, this must **not** be blocked by the local firewall in order to allow automated testing.

1.2 Starting QF-Test and Loading the Test-suite

After starting up QF-Test, you can immediately bring up our first example test-suite. You can locate the test-suite file by bringing up the open-file dialog window with the File→Open menu option. Select the directory `qftest-4.0.8/doc/tutorial` from your QF-Test installation. From this directory, select the file `Options.qft`. QF-Test will then load the indicated test-suite which should look as follows:

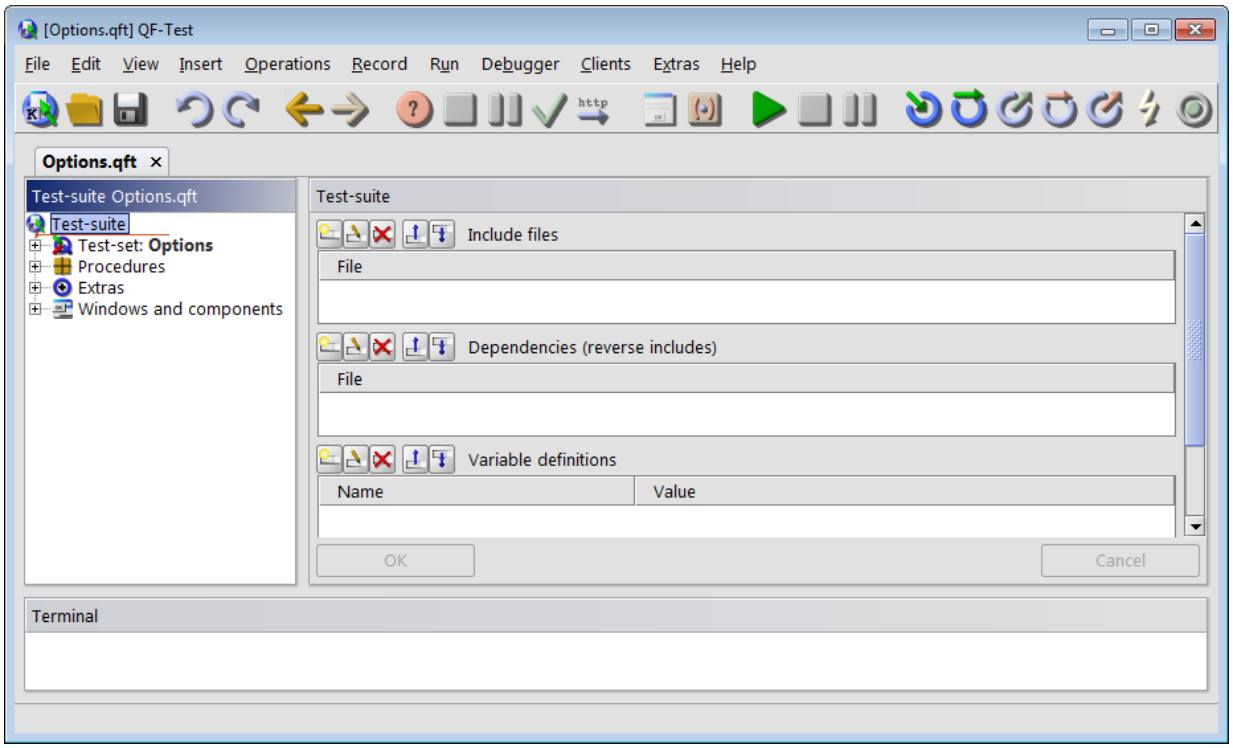

Figure 1.1: The Test-suite `Options.qft`

The left part of the main window contains the tree structure that represents the test-suite. The right side displays the details of a selected node. (In case you don't see the details view, you can switch it on with the View→Details menu option.) At the bottom of the main window you'll see the terminal output area, which displays standard messages and communications in between your test-suite and the client application you are testing. Displaying the terminal output can be toggled on or off using the View→Terminal→Show menu option.

With the help of the tree structure in the main window, you can navigate and select individual nodes of the test-suite. When a node is selected, you'll see its properties displayed in the details area on the right.

Double click on the node "Test-set: Options" to expand the nodes underneath it. You'll see that basically three test-cases are contained in this test-set: "Clickstream", "Text check" and "Selected test". Those test-case nodes are enclosed by a "Setup"/"Cleanup" pair to ensure a proper system as base to run the test-cases on.

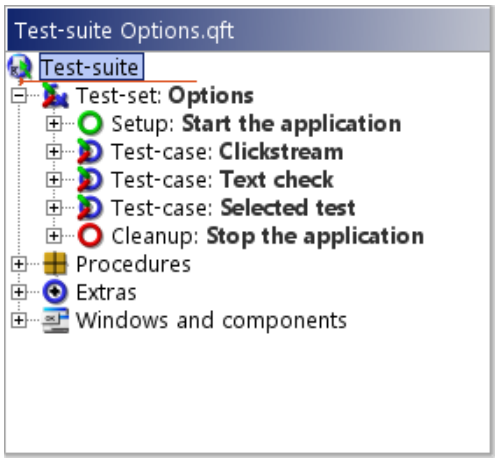

Figure 1.2: The Content of the Node ”Test-set: Options”

In the following sections we’ll describe the purpose and function of these individual nodes.

1.3 Starting the Application

Our first step will be to examine the ”Setup” node. Expand this node now, so that you see the contents as shown in the following diagram:

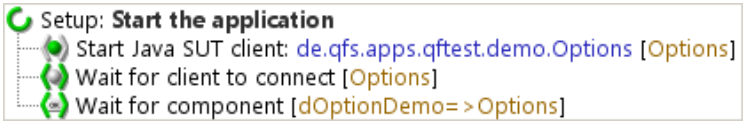
Setup: Start the application
Start Java SUT client: de.qfs.apps.qftest.demo.Options [Options]
Wait for client to connect [Options]
Wait for component [dOptionDemo=>Options]

Figure 1.3: The Setup Node

Within the ”Setup” node you’ll see three child-nodes:

- ”Start Java SUT client” - starts the application for the client you will be testing, called the System Under Test (SUT).
- ”Wait for client to connect” - waits until the new Java Virtual Machine, in which your SUT will run, binds itself to QF-Test.
- ”Wait for component” - waits for the appearance of a certain component of the SUT. For this example, we wait for the main window of the SUT itself. Don’t worry if this step isn’t completely clear at the moment, we’ll explain later about components and their meaning.

Now click on the ”Start Java SUT client” node so that it is selected as seen below:

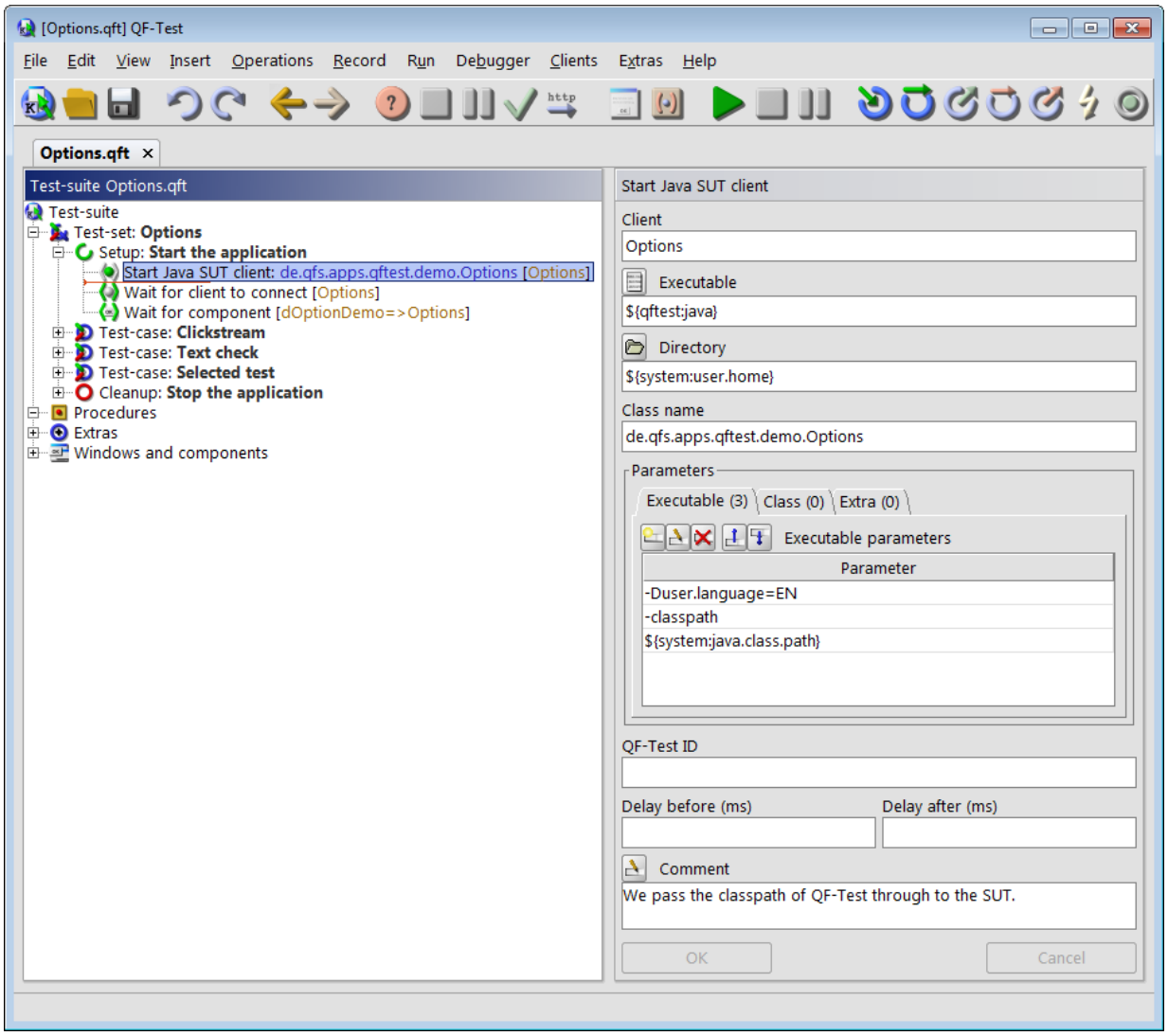

Figure 1.4: The Start Java SUT Client Node

Let’s take a closer look at the properties of this node, as displayed in the details window on the right side:

- In the first field, labeled ”Client,” is an unambiguous name for the SUT application. You can give the SUT any name you wish, but the same name will be used as a reference to the SUT in other nodes. In other words, this name must be used consistently.
- The ”Executable” is the second field, in which the executable name of the program is specified. For our example, the variable ”${qftest:java}” has been given. This

variable represents the Java program you established during the installation of QF-Test. You could alternatively manually enter the entire path to your Java Virtual Machine, along with the java executable. With the assistance of the ”Select executable file” button you can enter in an executable with a standard file-browser.

- The field ”Directory” indicates the working directory for the SUT when it is started.
- The fourth field, labeled ”Class name,” specifies the Java class used to implement the SUT. This class must contain the `main()` method, which will be called by Java when the SUT is started.
- Under ”Parameters” you see the command line arguments for the SUT. In this example we specify the argument to set the application language to English. In addition, the classpath is set appropriately, which is necessary because a class of QF-Test itself is used as SUT.
- The remaining property fields can be left empty and are not required for our example. You will note, however, that a comment has been filled in (the last field). All nodes have an optional comment field which you as a test-suite developer may find helpful for documenting a node’s purpose.

At this point, we’re ready to start the SUT. Click on the ”Setup” node again (so that it is selected), but still expanded (the child-nodes still visible). Now click on the ”Start test run” button . This button causes the selected node - ”Setup” in our case - to be executed.

When a node executes, QF-Test will use the tree structure to show you what is happening. An active node will be marked with an arrow pointer ”->” that indicates which node is being executed. You’ll also see messages from the SUT client in the terminal display.

When the ”Setup” sequence is completed, our demo application ”Options Demo” will appear on your screen. This is the SUT, fully under the control of the watchful eye of QF-Test.

Note

If the Options demo is not visible on your screen or only appears for a short moment, it is probably covered by the main window of QF-Test. The best solution is to place both windows side by side so you can watch them at the same time. In case you get an error dialog indicating that QF-Test is not able to connect to the SUT, please take a look at manual chapter 3.1.3.

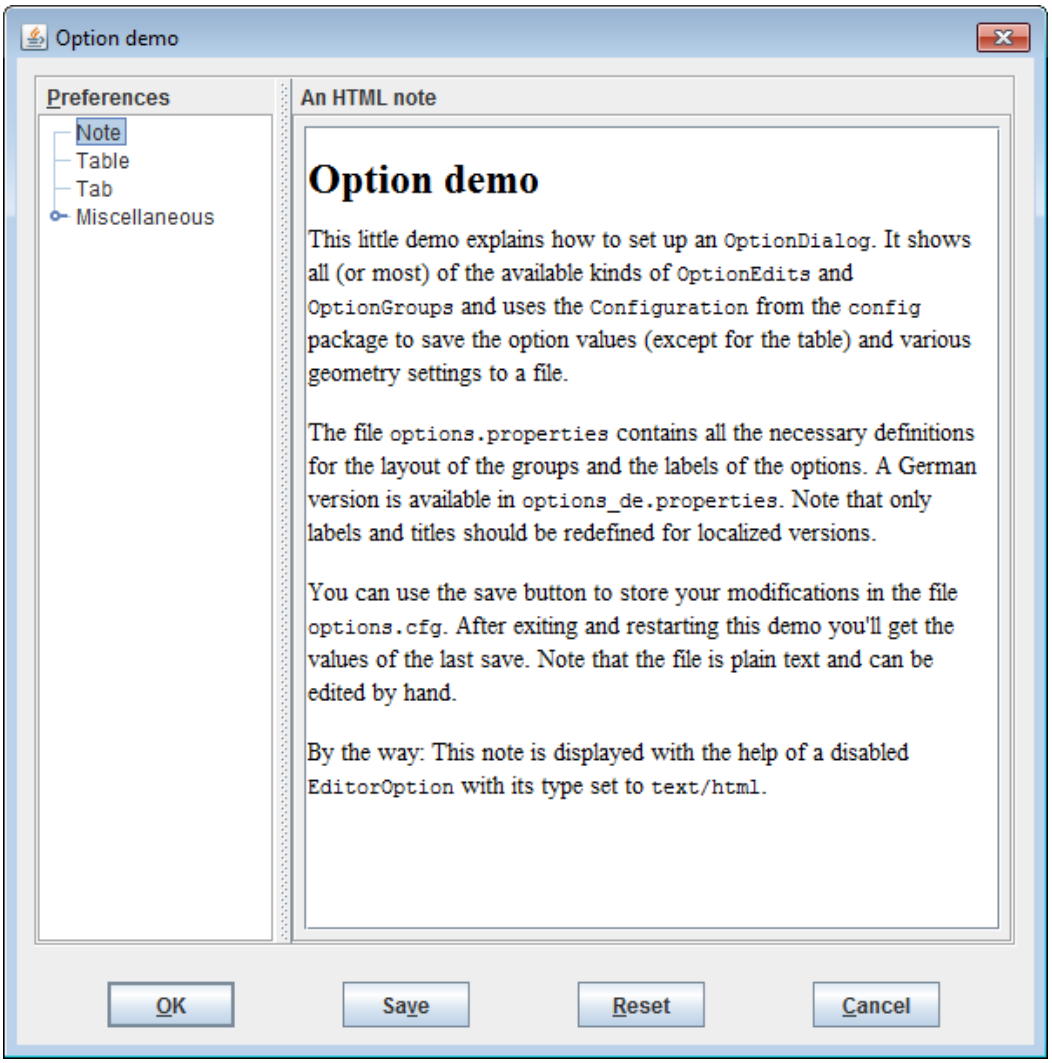
Option demo
Preferences
Note
Table
Tab
Miscellaneous
An HTML note
Option demo
This little demo explains how to set up an OptionDialog. It shows all (or most) of the available kinds of OptionEdits and OptionGroups and uses the Configuration from the config package to save the option values (except for the table) and various geometry settings to a file.
The file options.properties contains all the necessary definitions for the layout of the groups and the labels of the options. A German version is available in options_de.properties. Note that only labels and titles should be redefined for localized versions.
You can use the save button to store your modifications in the file options.cfg. After exiting and restarting this demo you'll get the values of the last save. Note that the file is plain text and can be edited by hand.
By the way: This note is displayed with the help of a disabled EditorOption with its type set to text/html.
OK
Save
Reset
Cancel

Figure 1.5: The Options Demo

1.4 Clickstream Test-case

Our next task is to take a first look at how a test-case for the SUT in QF-Test is structured. The test-case node labeled ”Clickstream” contains several sequences of operations that perform functions within the SUT, such as mouse-clicks and populating text-fields. Through this test we’ll demonstrate several simple but very important features of QF-Test.

When you expand the ”Clickstream” test-case node, you’ll see the two sequences as shown below:

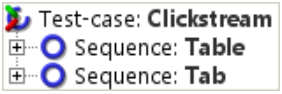

Figure 1.6: The Clickstream Test-case Node

To execute the test, select the ”Clickstream” test-case node and click the replay button . The sequences will then be carried out directly in the SUT, as you’ll see during execution and are supposed to pass through without problems.

The test result is indicated during and after the test run in the status line at the bottom of the QF-Test main window and says ”No errors”. Next to it there are counters for the numbers and results of the executed test-cases. In our case it was just one that performed error-free which means a success rate of 100%.

Hint: Let the mouse cursor hover above a counter icon to see its description. A listing of all possible counters can be found in chapter Capture and replay of the manual.

For a better understanding of what you saw during replay, let’s take a look at the sequence ”Table” within the ”Clickstream” test-case:

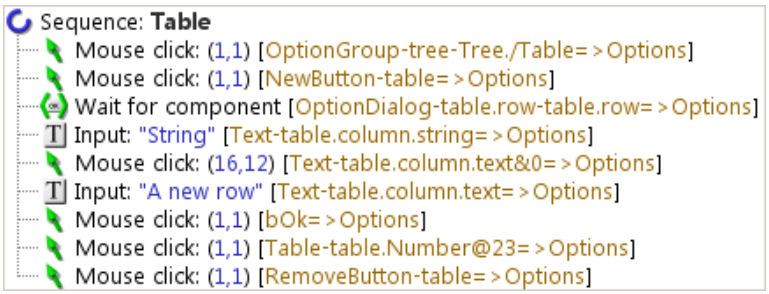

Figure 1.7: The ”Table” Test Sequence

Here we see that the sequence starts with several mouse clicks. Next to the ”Mouse click” label you’ll see the actual coordinates used to implement the mouse click relative to the component listed in the brackets next to the coordinates. The first node is, for example, a click on the element of the SUT labeled ”Table” in the tree-structure ”OptionsGroup-tree-Tree”. The meaning of the syntax in detail will be explained later.

After the first few mouse-clicks in the sequence there is a ”Wait for component” node that causes the execution of the sequence to wait for the appearance of the SUT’s edit dialog. After the dialog window appears, execution of the sequence will continue, in which the ”Input” node is activated to enter a string into a field in the dialog window. The remaining nodes of the sequence have similar functions to those mentioned above and should be comprehensible from the descriptions you see.

In case you're feeling curious, expand the other sequence ("Tab") too. Inside this sequence you'll find similar nodes to those you saw in "Table" with the addition of one node used to control the keyboard.

1.5 A Few Tips

We take a short break in this section to give a few tips that might prove helpful once you continue on with the tutorial.

While working with various nodes in QF-Test you may, for example, require some assistance in remembering what purpose they are used for. To serve this need, QF-Test comes with context-sensitive help, driven simply by your mouse. Move the mouse pointer over an element you would like to have assistance with, then click the right mouse-button. In the now appearing popup menu, you'll see an entry labeled "What's this?" By selecting this option from the popup menu, the appropriate section in the user's manual will be brought up and displayed in your standard browser. You'll hopefully find the manual quite helpful and complete in answering questions that may arise.

We should also mention that there is a PDF Version of the user's manual available in the installation directory of QF-Test. Within the PDF version you may find it easier (in comparison to the HTML version) to search for specific terms and concepts.

1.6 Text Check

One of the most important concepts in QF-Test is that of a check; that is, a query of certain elements in the SUT. A 'text check' queries the existence or absence of text within a component of the SUT, such as a text field.

For example, let's make a check of a field in the SUT client. Bring up the window of the "Options Demo" and click on the "Miscellaneous" node listed in the left-hand side tree structure so that the node expands to display its child nodes. Within the child nodes you'll see the node "Numbers." Click on this node, which will bring up on the right side a new window which includes the text field labeled "May be negative." For our example, we'll do a check of this field.

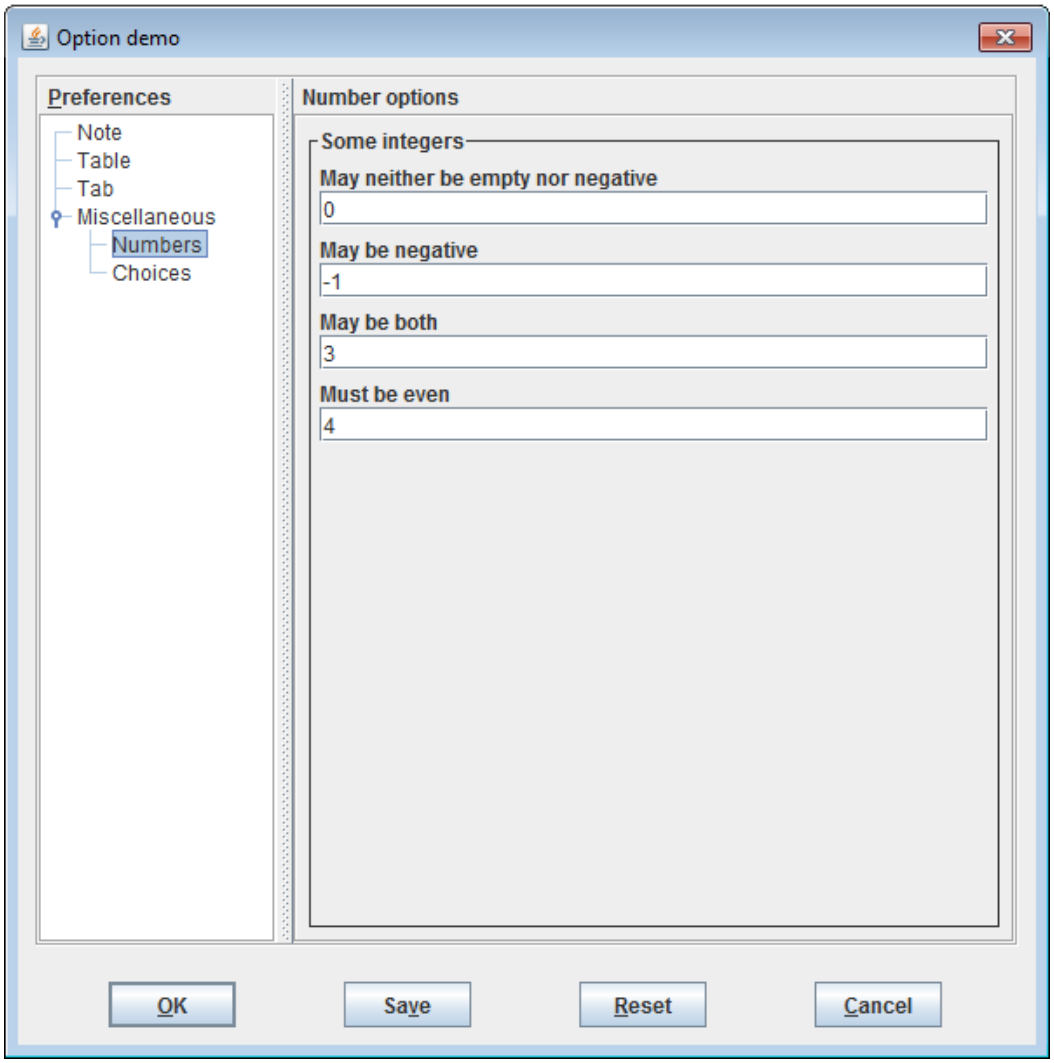

Figure 1.8: The Text Field to Check

Now return to your test-suite and expand the ”Text check” test-case node. Within you will see the following test sequence:

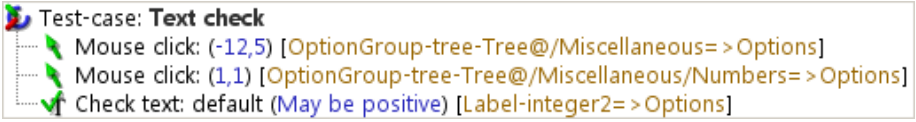
Test-case: Text check
Mouse click: (-12,5) [OptionGroup-tree-Tree@/Miscellaneous=>Options]
Mouse click: (1,1) [OptionGroup-tree-Tree@/Miscellaneous/Numbers=>Options]
Check text: default (May be positive) [Label-integer2=>Options]

Figure 1.9: The Text Check Test-case

It consists of two mouse clicks, aiming to open the ”Misc” node and select the node ”Numbers”, and a check of the label text.

With the ”Check text” test-case node selected, click on the replay button to execute it. When the sequence completes, a dialog-box will appear indicating that an error occurred:

Figure 1.10: Error in Check Text Node

What happened? Whenever such incidents occur, taking a look at the QF-Test run-log can prove very useful for diagnosis. You can open it directly via the ”Open run-log” button on the error dialog. Alternatively the latest run-log can be opened from the test-suite via the Run→1. ... menu item or by pressing Control-L. A new window will then appear displaying the recorded contents of the sequence you just executed:

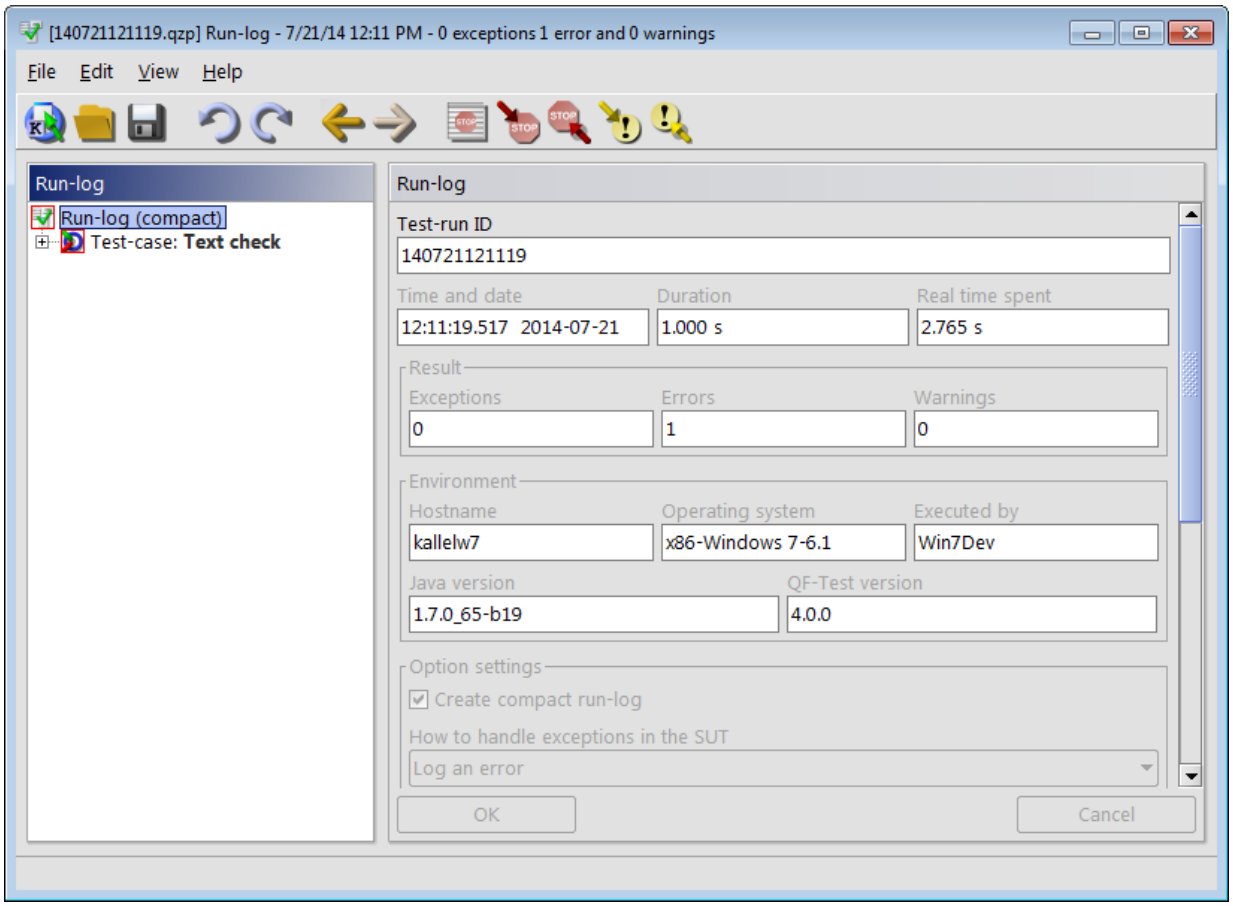

Figure 1.11: Run-Log for the Check Text Sequence

The run-log is similar in structure to the tree structure you are already familiar with from the normal test-suite view. The tree on the left side represents the time-recorded events of the test run, with the first event at the top, the last at the bottom. When you click on one of the event nodes on the left side, the properties of the event will be displayed on the right.

As you expand the nodes in the tree structure on the left, you'll notice immediately that some nodes are surrounded by a suspicious-looking red box. As you may suspect, this is an indication by QF-Test locating where a problem occurred in a child-node. If you keep expanding the red nodes, you'll eventually come to the actual problem node. An easier way to find the error is to simply use the Edit→Find next error menu option of the run-log's menu. This leads to the following:

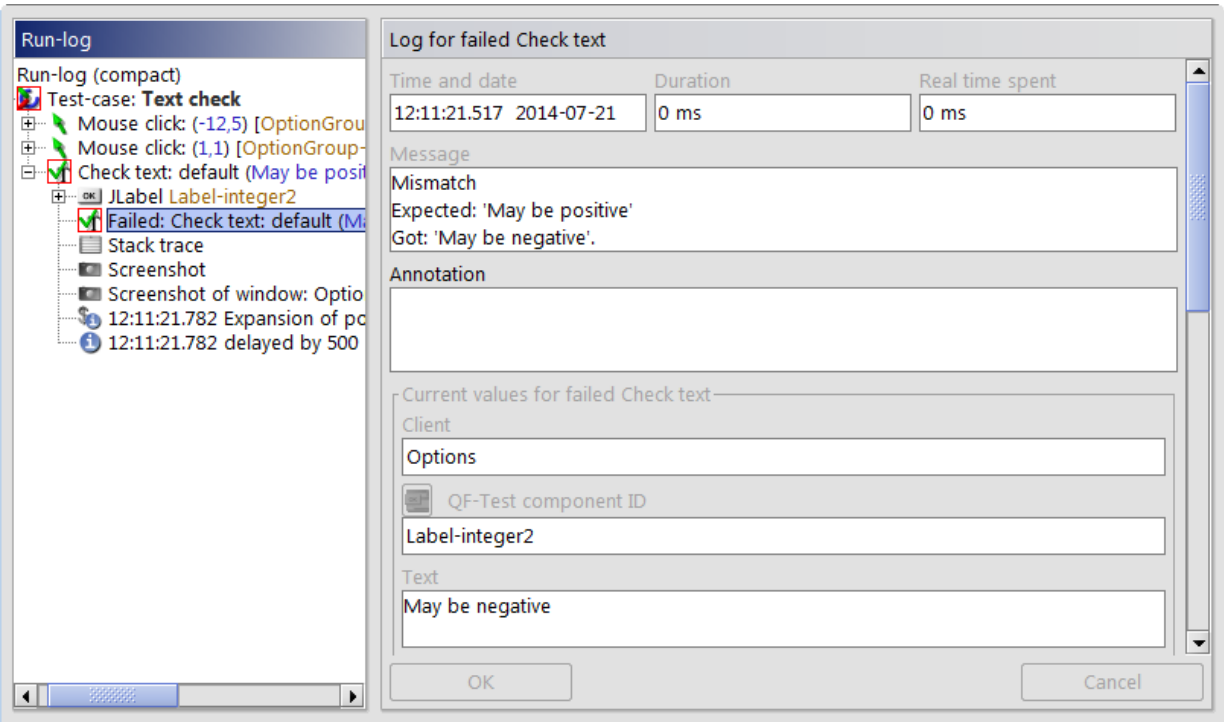

Figure 1.12: Diagnosis of Error in the Check Text Sequence.

When you click on the final red node, you'll see in the details of this node that there was a deviation between the text which was expected in our SUT field and the one which was actually found. The typical next step is to jump to the corresponding node in the test-suite and correct it. Provided that the appropriate node in the run-log is selected, you can use Edit→Find node in test-suite or simply type Control-T. One step beyond goes a functionality provided by the run-log node titled "Failed: Check text" in figure 1.12(13) available from the context menu, to be opened via a right click - Update check node with current data or Control-U.

Another useful feature for error diagnosis is the tree node labeled "screenshot" in the run-log. Its details contain a full screenshot taken at the time when the error happened. Being able to see the state of the SUT at that moment is a valuable aid in determining the cause of the error. The following image shows a screenshot node:

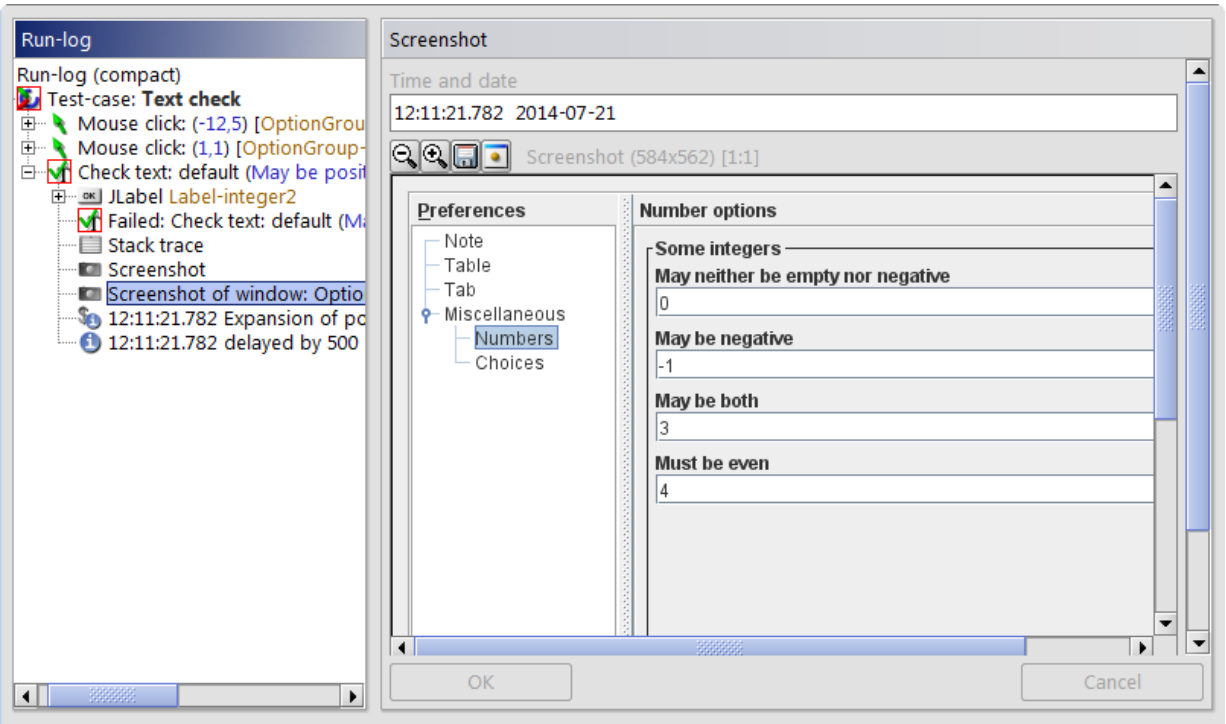

Figure 1.13: Screenshot node showing the error situation

Beside a screenshot of the whole screen, QF-Test also saves independent images of all SUT windows. For our example it is the run-log node ”Screenshot of window: Option Demo”. Such can be helpful when the SUT is hidden by another application window at the time when the error occurs.

Note: The information gathered from a long test-run accumulates and can devour enormous amounts of memory. Therefore QF-Test is typically configured to create a special form of compact run-log. Herein only the last 100 protocol nodes are kept, the rest is discarded, except for information relevant for report generation and for error diagnosis which is always retained. This functionality is configurable through the option ”Create compact run-log” within Edit→Options →Run-logs→Content. The type of a run-log is also shown in its root node. The number of screenshots stored in the run-log can be configured as well.

As a simple exercise, you may wish to modify the expected text in the ”Check text” node of your test-suite. Now when you execute the sequence, no error will occur. Please reverse the text to its previous state by Edit→Undo or Control-Z once the test has finished, because we will need the error for demonstration purpose later on.

1.7 Checking a Radio Button

The third test sequence ”Selected test” performs queries about the state of a specific radio button in the SUT in order to determine if that radio button is selected. Under the ”Choices” section of the ”Miscellaneous” area in the ”Options Demo” application, you can interactively select any one of four radio buttons. We’ll use these radio buttons to implement a test that includes a successful check as well as a check that leads to an error.

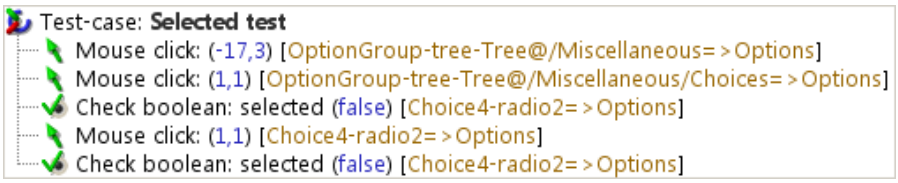
Test-case: Selected test
Mouse click: (-17,3) [OptionGroup-tree-Tree@/Miscellaneous=>Options]
Mouse click: (1,1) [OptionGroup-tree-Tree@/Miscellaneous/Choices=>Options]
Check boolean: selected (false) [Choice4-radio2=>Options]
Mouse click: (1,1) [Choice4-radio2=>Options]
Check boolean: selected (false) [Choice4-radio2=>Options]

Figure 1.14: The ”Selected test” Node

The node ”Selected test” contains three mouse click events and two checks. The first two clicks cause the nodes in the tree structure of the SUT to be expanded until the ”Choices” node is visible and its properties displayed on the right. Afterwards, you see a ”Check boolean: selected” node that queries the state of the fourth radio button to determine that it is not selected. The third mouse event then performs a click on this very same fourth radio so that it is now selected. Finally, another ”Check boolean: selected” is performed just as before. However, now that the radio button is selected, you’ll see that this last check leads to an error.

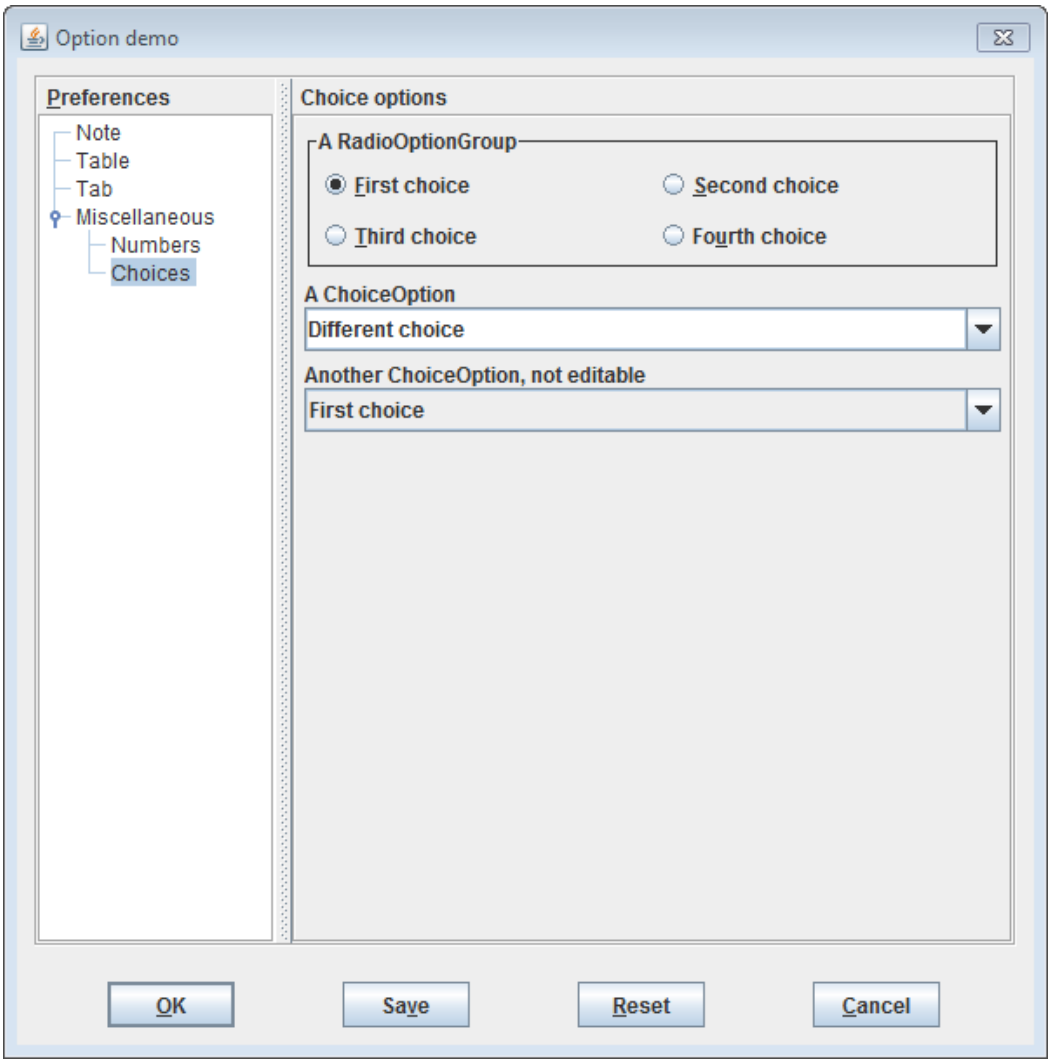

Figure 1.15: Radio Buttons in the Options Demo

Try running the ”Selected test” sequence now so that you can see the actual execution of the description we just provided. After running the test-sequence, you may want to collapse the nodes in the SUT so that it is back in its original state (the ”Miscellaneous” child-nodes no longer visible).

1.8 Stopping the Application

The function of the ”Cleanup” sequence is important to understand. Inside a test-set a ”Cleanup” node will be executed after every test-case located in the same level of the tree-structure, just as the ”Setup” sequence will be executed before each test. The

”Cleanup” node serves the purpose of leaving the SUT in a specific or ’clean’ state so that further tests can be executed deterministically. This step is equivalent to the ”tear-down” method within unit testing.

Note however that the ”Cleanup” node, just like the ”Setup” node, will only be executed when the top-level node is executed or when you explicitly execute it. In the previous sections, you specifically selected and carried out one specific test or sequence (such as ”Clickstream”), which did not lead to the ”Setup” or ”Cleanup” nodes being run. In the next section, however, you’ll see visually how this all works when you execute the top-level ”Test: Options” node.

Expand the ”Cleanup” node now to take a look at its child-nodes. For this ”Cleanup” node, we attempt to stop and close the SUT application in an orderly fashion by clicking the ”Cancel” button of the SUT. In case this desired result does not occur (the SUT does not close), a more aggressive attempt is made.

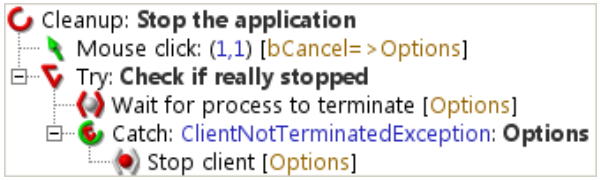
Cleanup: Stop the application
Mouse click: (1,1) [bCancel=>Options]
Try: Check if really stopped
Wait for process to terminate [Options]
Catch: ClientNotTerminatedException: Options
Stop client [Options]

Figure 1.16: The Cleanup Sequence

Our ”Cleanup” sequence is organized as follows:

- The first node clicks the ”Cancel” button of the SUT.
- The remaining nodes are built around the frame of a try/catch construct, which is used to handle the erroneous case of the SUT failing to stop when ”Cancel” is clicked. The first node in the try/catch block performs a ”Wait for client to terminate” check.
- In the case where the SUT client fails to terminate within the given timeout period, an exception is thrown - called ”ClientNotTerminatedException” - which will then be caught within the ”Catch” node.
- If the exception is indeed thrown, a ”Stop client” node is then executed under the ”Catch” node. The ”Stop client” node causes the actual process running the SUT application to be terminated.

1.9 The Complete Test

In the previous sections we've taken you step-by-step through many elements of our example test-suite. At this point, you're now ready to execute the entire test in one fell swoop.

First, close the "Options Demo" in case it is still running. You can do this by executing the "Cleanup" node described in the previous section. Then select the "Test-set: Options" node and execute it with the replay button. The entire suite of tests will last a few minutes, due to the delays we built into certain nodes of the test-suite so that you can better follow what's going on. If you look in the details of the "Test-set: Options" node, you'll see a variable called "delay" defined in the parameter default values section under "Defaults". You can change this value if you wish to shorten or lengthen the total run time.

When the tests complete, you should end up with two failures, both of which we discussed in the previous sections. Open up the run-log again to take a closer look:

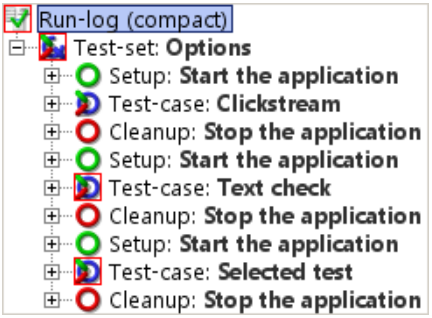

Figure 1.17: Run-Log for the Completed Test-set "Options"

Here you'll see in a visible fashion the process we attempted to explain in the last section concerning the behavior of the test-set node in which execution of the "Setup" and "Cleanup" nodes take place before and after each test. Thus a clearly defined "clean" state is ensured.

1.10 Report Generation

In the world of quality assurance the documentation and archiving of test results are of vital importance. In order to achieve this, QF-Test offers an automated report-generation feature. Since you've just finished a complete test run, we're at a good point to show you the capabilities of this feature.

Make sure the run-log for your test-run is open. Now use the File→Create HTML/XML report... option from the run-log's menu to bring up the dialog window used to specify the nature of your desired report.

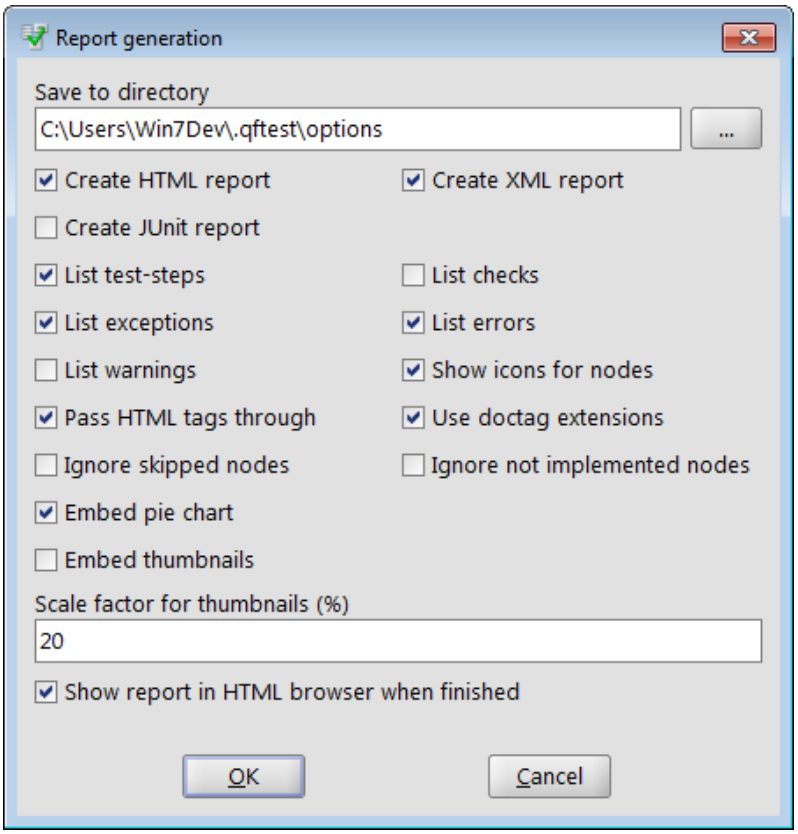

Figure 1.18: Report Generation Properties

In the first field, you can specify the file-name of the report. Following this, you can decide what type of report you want. QF-Test offers two kinds of reports, HTML or XML format. An XML report is useful if you've written your own XSLT stylesheets to shape the report into the style you'd prefer.

There are further options to control handling of HTML and doctags used inside comment fields as well as viewing the report in your browser after creation. You can just leave those fields unchanged. Further details can be found in chapter Reports and test documentation of the user manual.

Let's try generating a simple HTML report from the results of the last test-run. After possibly adapting the fields we indicated above and confirming the generation with the "OK" button the report will then be generated and generally look as follows in your browser:

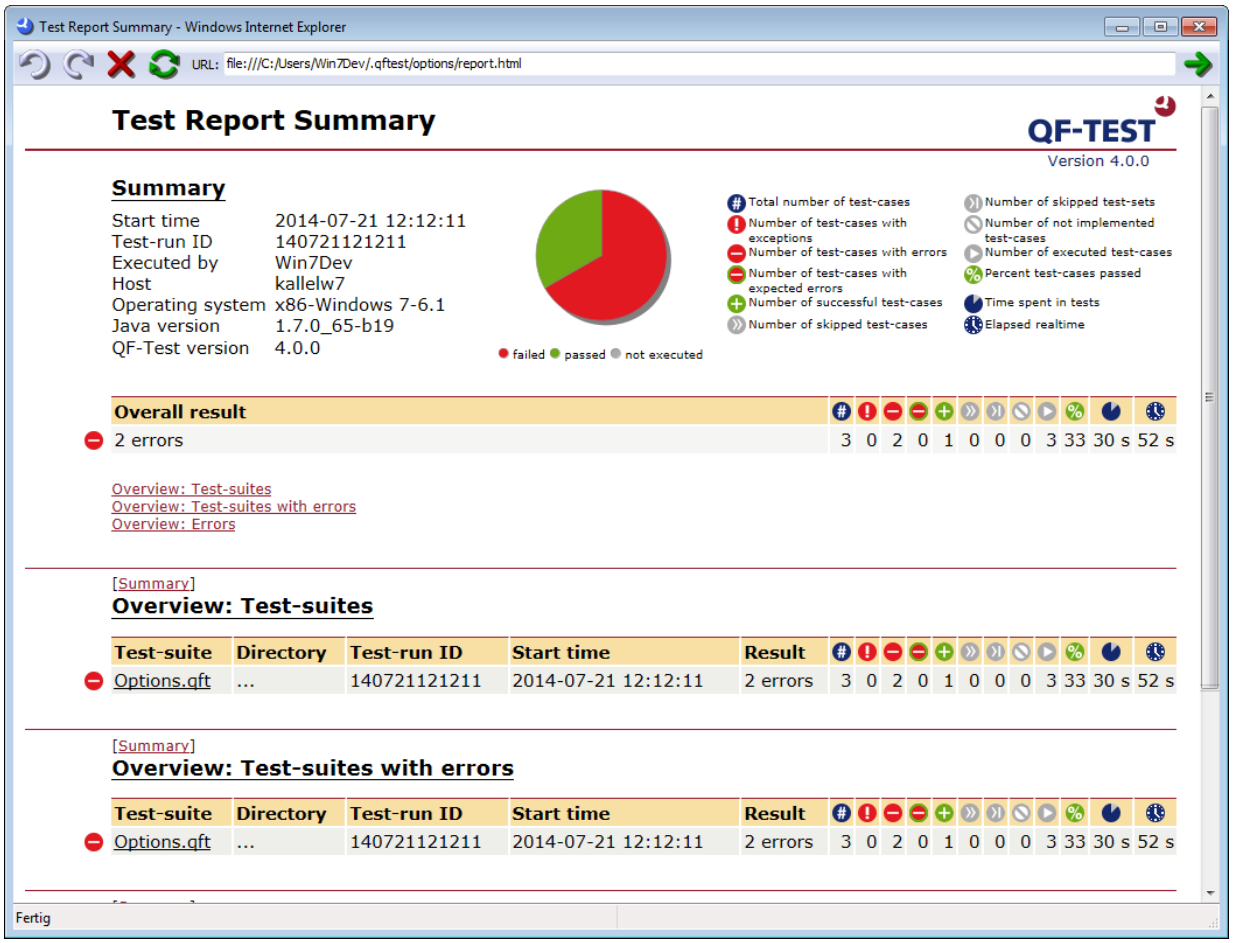
Test Report Summary - Windows Internet Explorer
URL: file:///C:/Users/Win7Dev/.qftest/options/report.html
Test Report Summary
QF-TEST
Version 4.0.0
Summary
Start time 2014-07-21 12:12:11
Test-run ID 140721121211
Executed by Win7Dev
Host kallelw7
Operating system x86-Windows 7-6.1
Java version 1.7.0_65-b19
QF-Test version 4.0.0
failed passed not executed
Total number of test-cases
Number of test-cases with exceptions
Number of test-cases with errors
Number of test-cases with expected errors
Number of successful test-cases
Number of skipped test-cases
Number of skipped test-sets
Number of not implemented test-cases
Number of executed test-cases
Percent test-cases passed
Time spent in tests
Elapsed realtime
Overall result
2 errors 3 0 2 0 1 0 0 0 3 33 30 s 52 s
Overview: Test-suites
Overview: Test-suites with errors
Overview: Errors
[Summary]
Overview: Test-suites
Test-suite Directory Test-run ID Start time Result
Options.qft ... 140721121211 2014-07-21 12:12:11 2 errors 3 0 2 0 1 0 0 0 3 33 30 s 52 s
[Summary]
Overview: Test-suites with errors
Test-suite Directory Test-run ID Start time Result
Options.qft ... 140721121211 2014-07-21 12:12:11 2 errors 3 0 2 0 1 0 0 0 3 33 30 s 52 s
Fertig

Figure 1.19: An HTML Report

The report begins with a summary containing informational data from your system on the top left side, a legend describing the meaning of icons used in the report on the top right side and the overall test result below. In our case as result we see the two well-known errors again.

Following the summary, an error overview lists the errors with its location where they occurred (the test-case) and the message describing the error details. Part three is a test-set overview just containing the ”Options” test-set, as our test-suite only contains that one. Finally all test-cases contained in the ”Options” test-set are listed with details like description, result and execution time.

The report generation feature proves very useful for creating an overview over the test-run and a document for presentation and archiving purposes.

Chapter 2

Creating a Test-suite [45-60 min]

2.1 Introduction

This chapter will take you through the steps of creating your first test-suite. The application to test will be the FileChooserDemo included in the Java software development kit. It is also included in the distribution of QF-Test together with the source-code, in the directory `qftest-4.0.8/doc/tutorial/FileChooserDemo`.

The FileChooserDemo application is a very nice demonstration of how you can customize the FileChooser class in Swing, to create your own ”Open”, ”Save” and customized file dialog for your application. We will use it as the SUT (System Under Test) for the creation of your own test-suite.

This chapter of the tutorial has been designed to illustrate the following concepts:

- Starting an SUT in QF-Test
- Recording components and organizing simple tests
- Creation of tests which check the dependencies of functions (business logic) within the SUT.

2.2 Starting the Application

Open a new (empty) test-suite from QF-Test with the File→New Suite menu option. Please be aware the ”old” test-suite remains open as a different tab.

Make sure that the detailed view is turned on, it can be toggled with the View→Show Details menu option. You now see the divided window with a tree on the left-hand side and a pane on the right-hand side representing the details of a selected

node. Using the tree you can navigate through the test-suite's nodes; when you select a node, you'll see its properties appear within the details pane on the right.

To begin you need to start the application. As we mentioned, our application to test will be the FileChooserDemo from the Java software development kit. The QF-Test Quickstart Wizard will help us to create an appropriate setup sequence.

Please open the Quickstart Wizard via the Extras menu or by pressing the record button in case it still shows the question mark. The Wizard should welcome you as shown in the figure below. After saying a short hello please press the "Next" button.

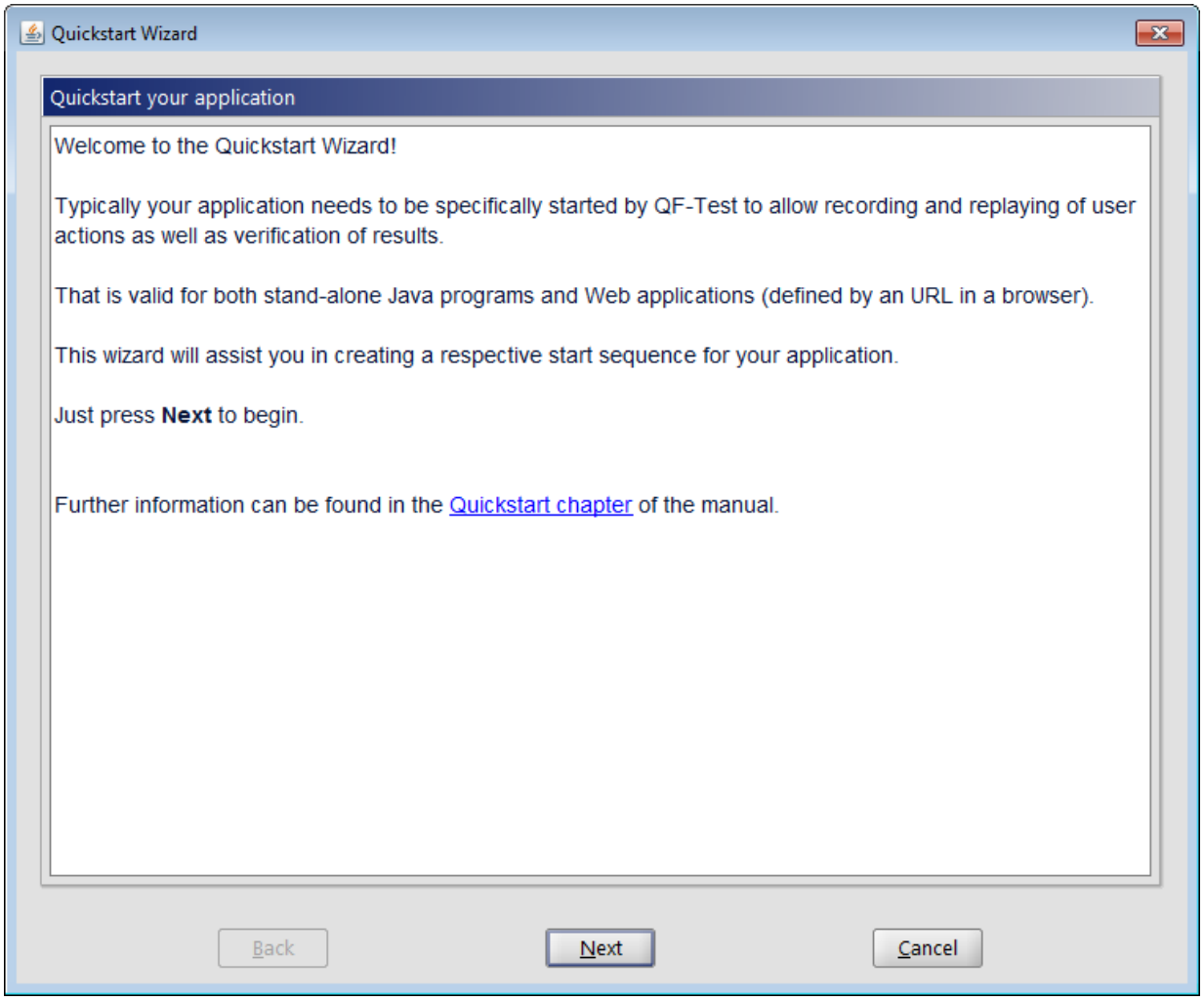
Quickstart Wizard
Quickstart your application
Welcome to the Quickstart Wizard!
Typically your application needs to be specifically started by QF-Test to allow recording and replaying of user actions as well as verification of results.
That is valid for both stand-alone Java programs and Web applications (defined by an URL in a browser).
This wizard will assist you in creating a respective start sequence for your application.
Just press Next to begin.
Further information can be found in the Quickstart chapter of the manual.
Back
Next
Cancel

Figure 2.1: The Quickstart Wizard

Next you can choose the type of application to be tested. Please select "A Java archive (`java -jar <archive>`)" and continue.

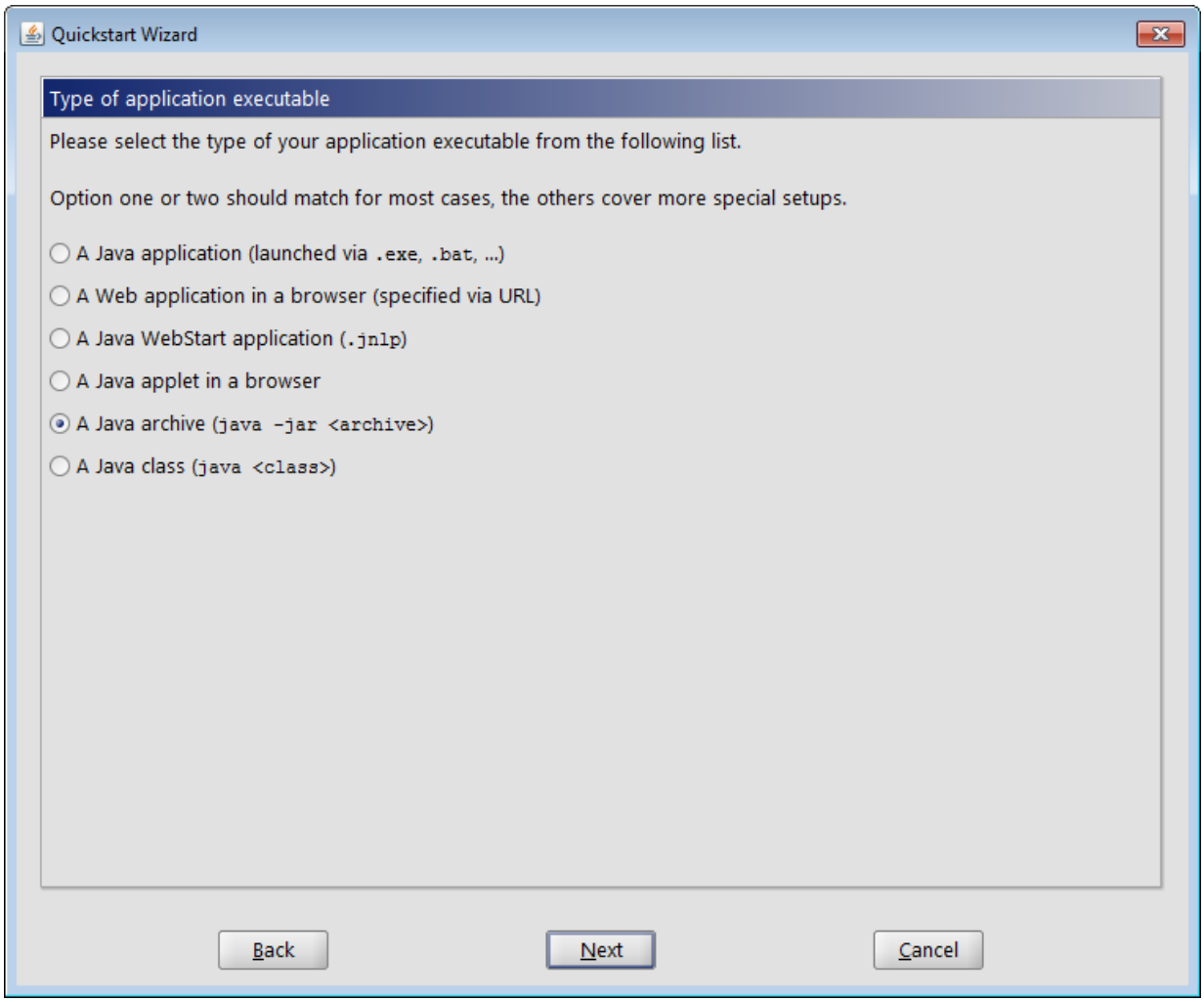
Quickstart Wizard
Type of application executable
Please select the type of your application executable from the following list.
Option one or two should match for most cases, the others cover more special setups.
A Java application (launched via .exe, .bat, ...)
A Web application in a browser (specified via URL)
A Java WebStart application (.jnlp)
A Java applet in a browser
A Java archive (java -jar <archive>)
A Java class (java <class>)
Back
Next
Cancel

Figure 2.2: Select type of SUT

Now you are asked about the Java executable. You can leave the proposed value `${qftest:java}` untouched which means the java installation QF-Test itself uses. Simply proceed to the next wizard step.

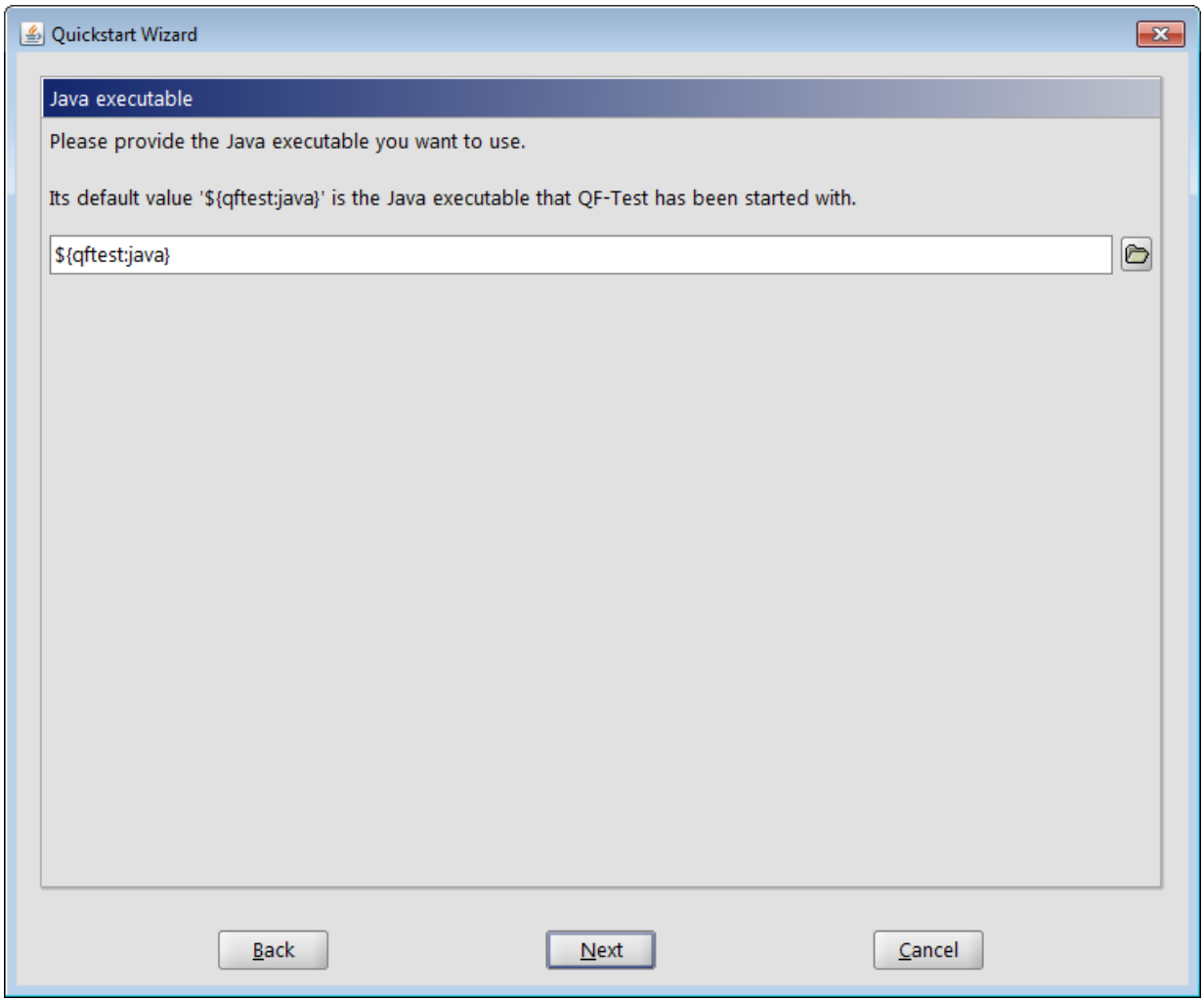

Figure 2.3: Specification of the Java executable.

On the next pane you are asked for an optional working directory for the application. As such is not required for our demo you can simply proceed to the next wizard step.

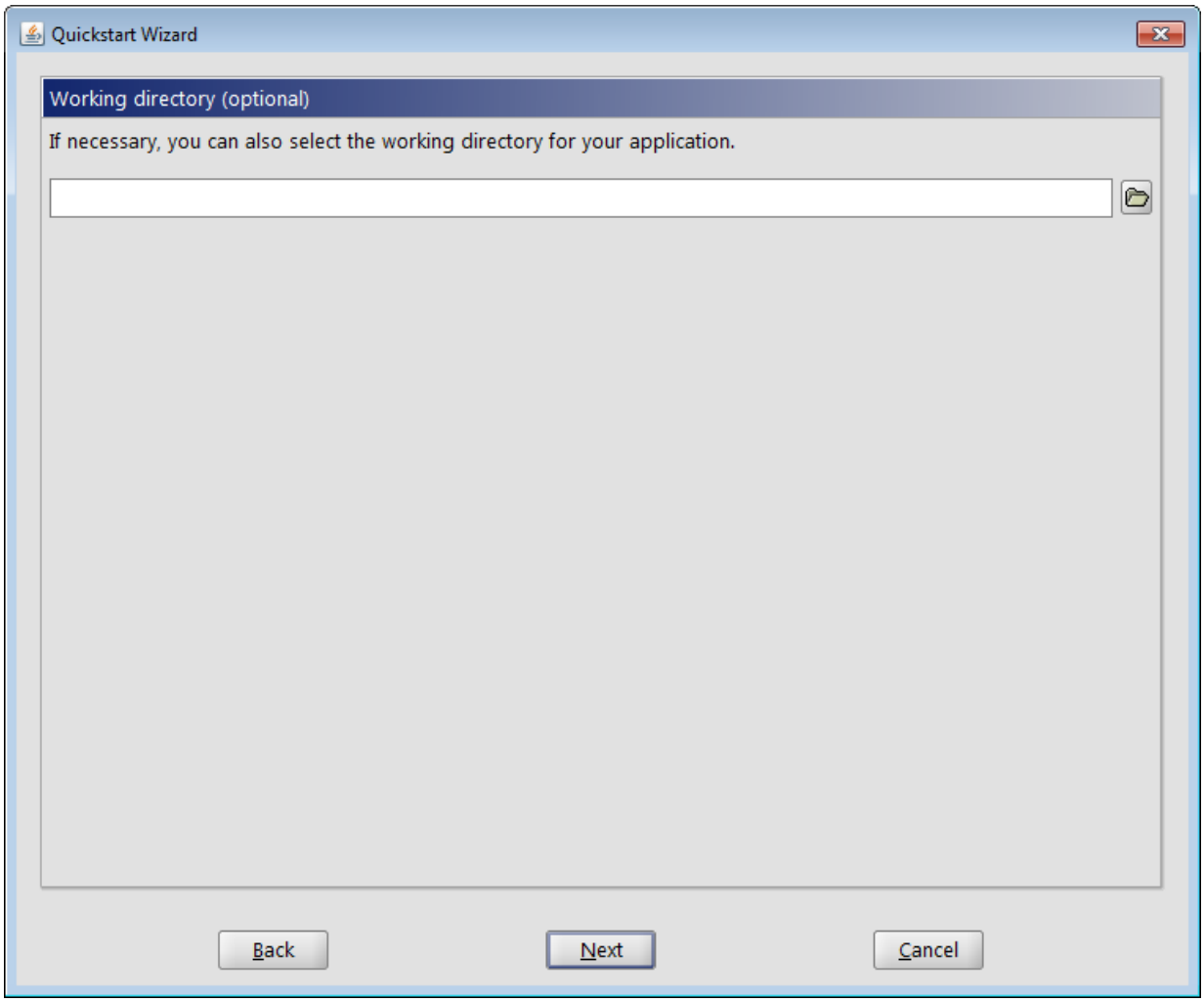

Figure 2.4: Working directory.

It's time now to select the Java archive to use. For our example it will be the `FileChooserDemo.jar` archive in the `doc/tutorial/FileChooserDemo` directory of your QF-Test installation. You can pick the file by using the "Select jar file" button or directly type into the text field. Another option (which works both on Windows and Unix) is to fill in `${qftest:dir.version}/doc/tutorial/FileChooserDemo/FileChooserDemo.jar` where the first part is a variable that will expand to the version specific installation directory of QF-Test.

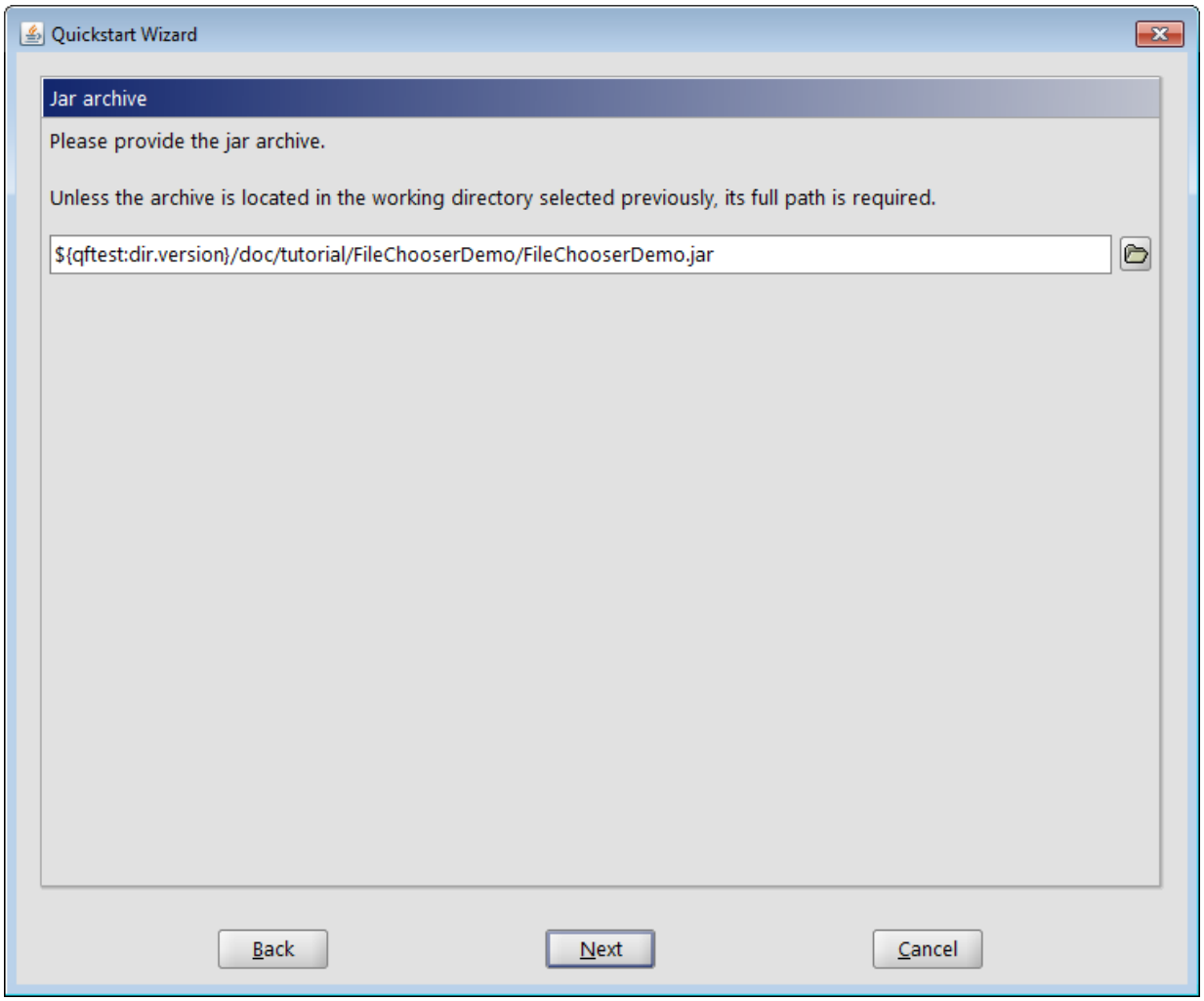

Figure 2.5: Jar file selection.

The next step of an optional SWT instrumentation can be simply skipped by ”Next” as our `FileChooserDemo` is not based SWT but on Swing.

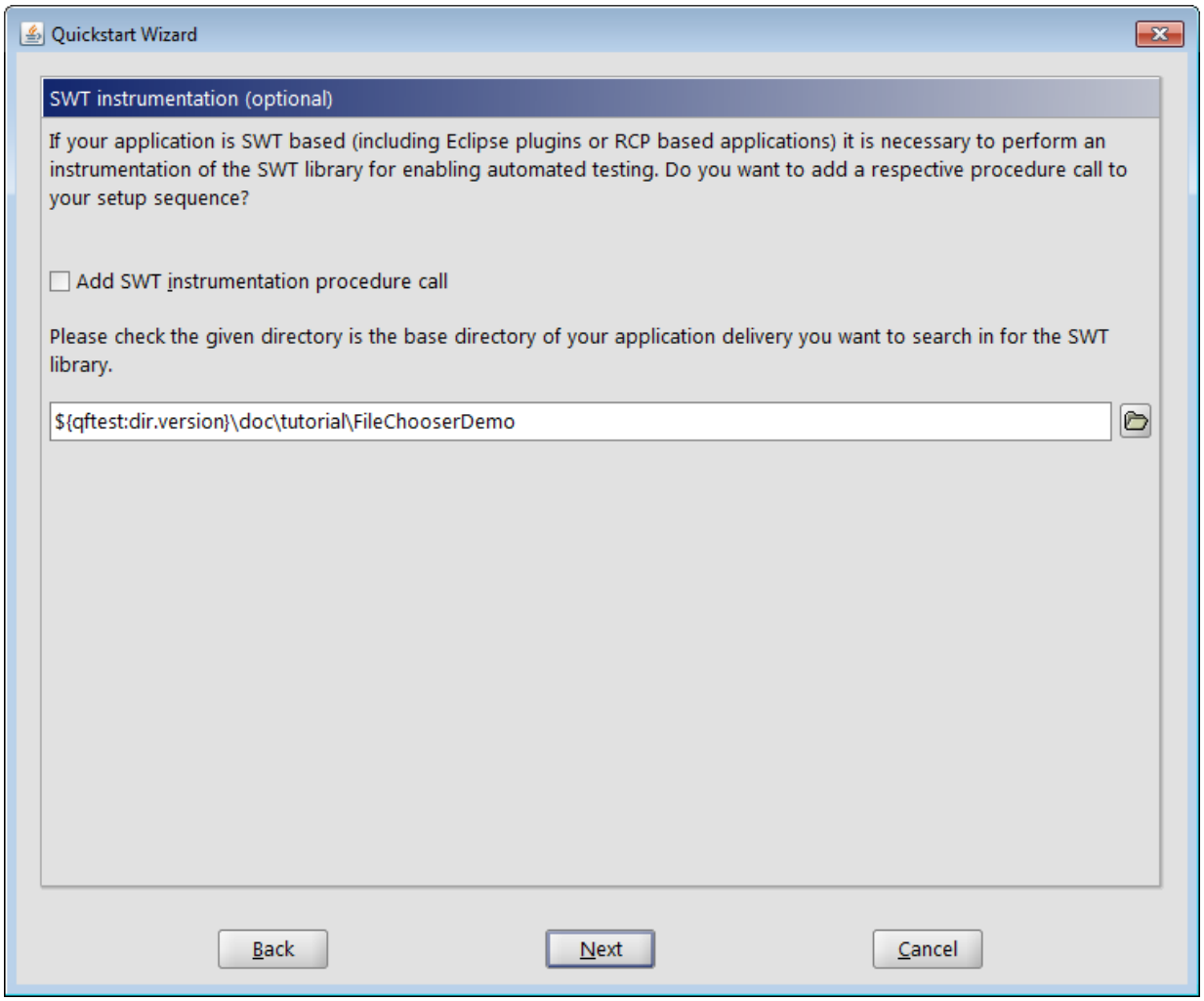
Quickstart Wizard
SWT instrumentation (optional)
If your application is SWT based (including Eclipse plugins or RCP based applications) it is necessary to perform an instrumentation of the SWT library for enabling automated testing. Do you want to add a respective procedure call to your setup sequence?
Add SWT instrumentation procedure call
Please check the given directory is the base directory of your application delivery you want to search in for the SWT library.
${qftest:dir.version}\doc\tutorial\FileChooserDemo
Back
Next
Cancel

Figure 2.6: SWT instrumentation

It's nearly done. We have just to assign a name to our client as a reference. Not quite unexpected let's call it "FileChooserDemo".

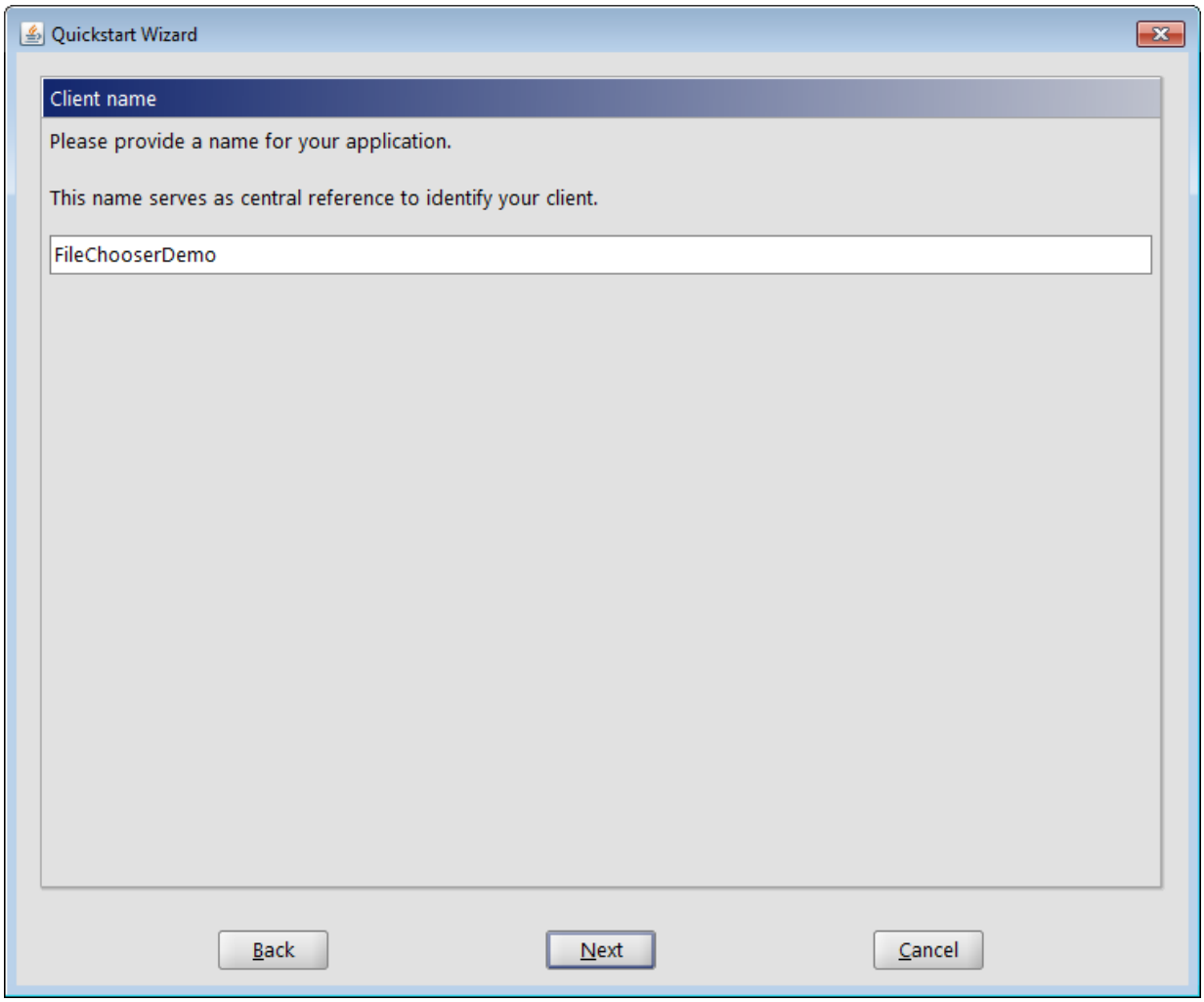

Figure 2.7: Client name

Finally some information is given about what to expect when the wizard finishes its task of creating a setup sequence and where to find help in case of troubles. Please disable the ”Start automatically” option, as we first want to take a look at the generated setup sequence and start it by hand. Now please press ”Finish”.

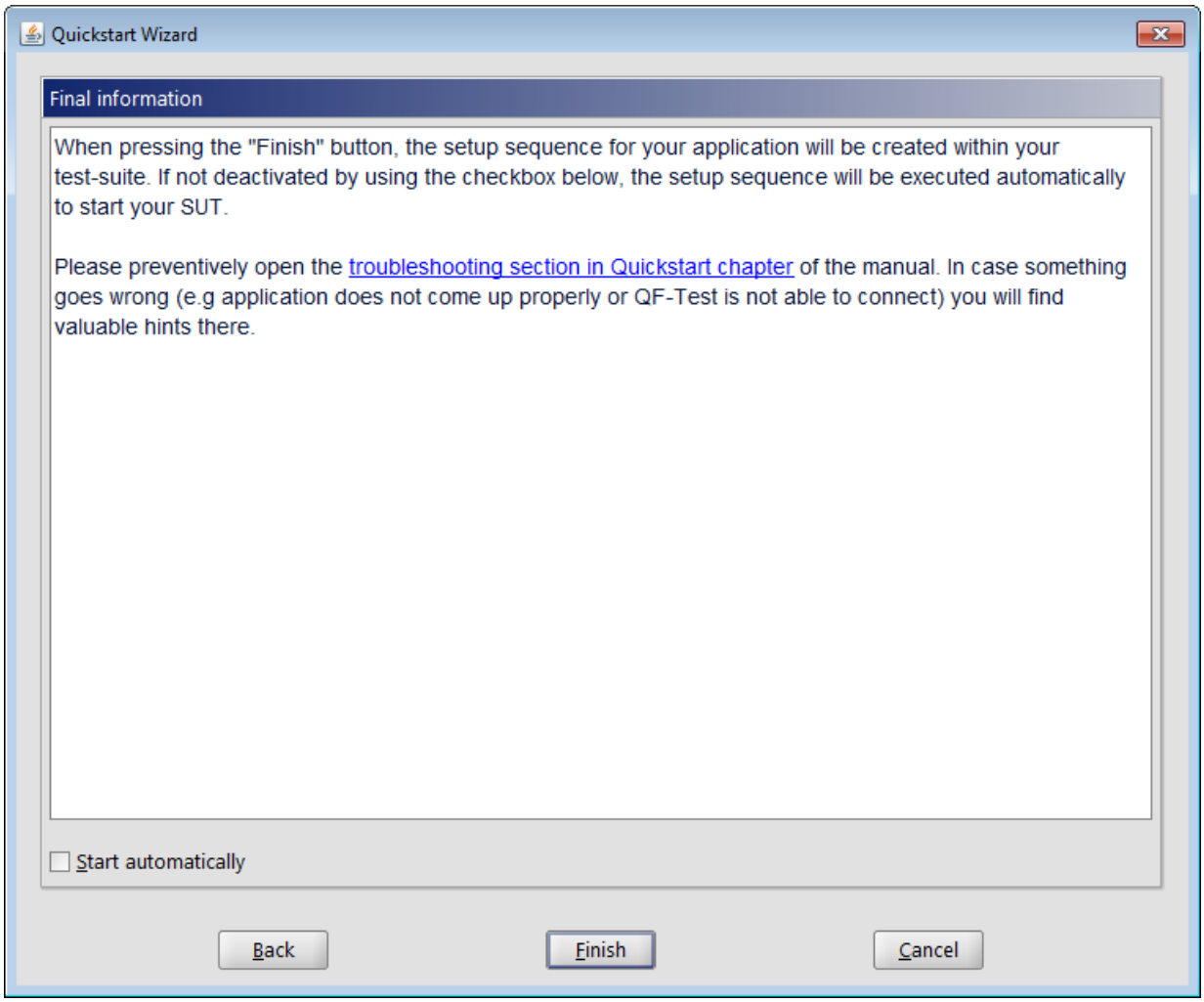
Quickstart Wizard
Final information
When pressing the "Finish" button, the setup sequence for your application will be created within your test-suite. If not deactivated by using the checkbox below, the setup sequence will be executed automatically to start your SUT.
Please preventively open the troubleshooting section in Quickstart chapter of the manual. In case something goes wrong (e.g application does not come up properly or QF-Test is not able to connect) you will find valuable hints there.
Start automatically
Back
Finish
Cancel

Figure 2.8: Final Information

The generated setup sequence appears under the ”Extras” and contains three steps:

- Set the global client variable
- Start the Java SUT
- Wait for the client to connect

Extras
Setup: Start Java application in a Jar archive
Set variable: client=FileChooserDemo
Start Java SUT client [$(client)]
Wait for client to connect [$(client)]

Figure 2.9: Generated Setup Sequence

You can have a look at the details of the nodes if you like. Basically you will find the values you provided during the Wizard steps assigned to the respective node attributes.

Now we want to see some action. Please ensure the ”Setup” node is selected. Then click the replay button or simply hit ”Enter”. You should see the FileChooserDemo application appear on your screen soon.

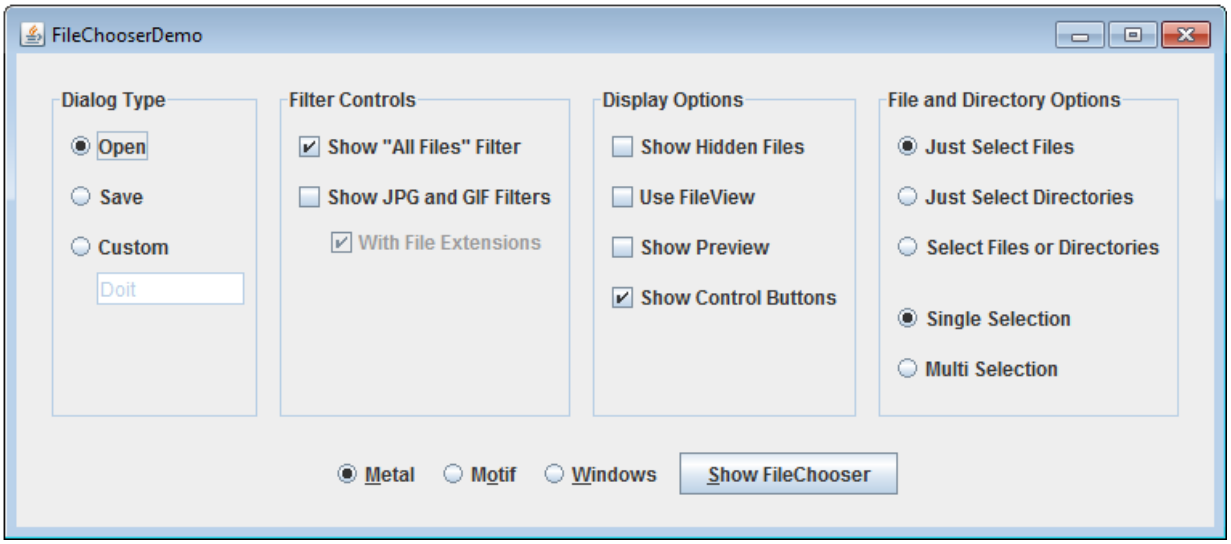

Figure 2.10: The FileChooserDemo Window

If the FileChooserDemo fails to come up after a few seconds, take a look at the terminal output display, which typically shows helpful messages.

2.3 Adding a Clickstream Test

You’re now ready to add a simple clickstream test. Press the ”Start recording” button and switch to the application window of the SUT. From now on every mouse and keyboard action performed within the SUT window will be recorded. Click on some buttons and switch back to your QF-Test test-suite window. Press ”Stop recording”

. You'll find the recorded sequence placed under "Extras" node on the left side of the main window, as seen below:

Figure 2.11: The Recorded Clickstream

The name of the recorded sequence will be entered in a standard fashion by QF-Test as the time and date of the recording. You can change this name as you see fit by simply clicking on the node and changing its properties in the details view on the right.

Select your recorded sequence now and execute it using the button. You should now see your exact sequence of mouse and keyboard events replayed within the window of the SUT. (Of course you may need to restart the SUT to wipe the slate clean.)

This type of test can be quite spectacular and can be used for demonstrations of your application, but it mostly tests the underlying Swing implementation and not the business logic of your application. In the last section of this chapter we will implement tests that check the content of fields and also check for causal connections within the FileChooserDemo (business logic), but first we will structure the recorded sequences into a test-suite.

2.4 Building a Test-suite

The basic structure of a test-suite below the 'root' node is fixed through the following nodes:

- An arbitrary number of "Test-set" and "Test-case" nodes to specify and structure functional tests.
- "Procedures" contains reusable sequences that can be organized into modular functions.
- "Extras" is the scratch pad for experimentation.
- "Windows and components" contain the all-important elements of the test-suite: registered elements of your SUT such as windows, menus and buttons. Each element within the "Windows and components" section contains properties of the

element so that QF-Test can find the component easily when replaying a test or sequence.

Functional test-cases are represented by ”Test-case” nodes and can be grouped and structured with help of ”Test-set” nodes. ”Setup” and ”Cleanup” nodes are intended to contain the steps to ensure a well-defined state before and after a test-case.

One possible implementation of the ”Setup”/”Cleanup” pair is to start and stop the SUT application; this way the SUT is always in a known state when a test runs. This pairing can, however, be used at any level and for any purpose you see necessary, like opening and closing a dialog, for example.

Let’s start by renaming our top-level test-set node from ”unnamed” to ”FileChooser Tests”. A dialog is popping up asking whether to update references, which we can simply confirm with ”Yes”.

The second step is to move the ”Setup” node generated by the Quickstart Wizard from the ”Extras” node into the ”Test-set” node - on the first position preceding to the ”Test-case” node. Moving the ”Setup” node can now be accomplished with mouse (Drag&Drop), context-menu (right mouse-button copy/paste) or with the Control-X and Control-V keyboard commands.

Note When using Drag&Drop a target node can be expanded by having the mouse cursor hovered above the ”+” left of the target node.

You might want to assign a more specific description to the setup sequence like ”Start FileChooserDemo”.

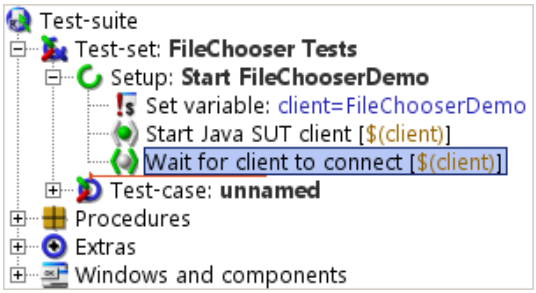

Figure 2.12: Start organizing the Test-suite

When this is done, add a ”Wait for client to connect” node with the Insert→Process nodes→Wait for client menu option. This node causes QF-Test to wait for the SUT client getting started and ensure that you get a successful connection to the SUT.

We need one more thing before the ”Setup” node is complete. After the SUT has connected to QF-Test it can take quite a while until the first window is shown on the screen.

We must make sure that the actual test doesn't start before that so we will need to wait for that window to appear.

To that end insert a "Wait for component" node after the "Wait for client" node with the Insert→Miscellaneous→Wait for component menu option. If the SUT is running, the "Client" attribute should already be filled in with the name of the SUT client, "$(client)" in our case. To set the QF-Test component ID, click on the button above the field to bring up a dialog with the available "Window" and "Component" nodes. Select the "Window" node labeled "JFrame frameFileChooserDemo".

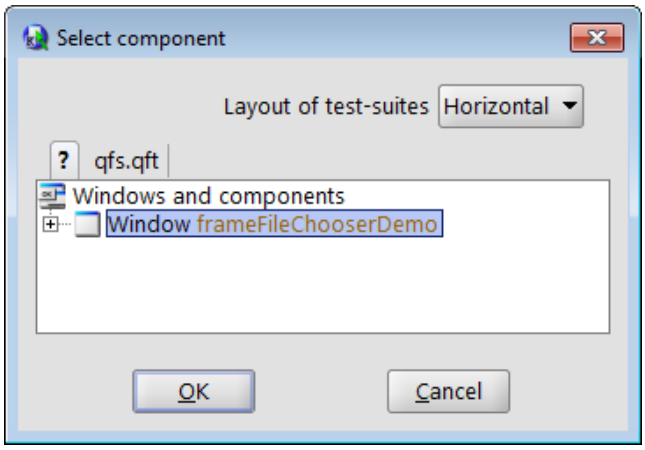

Figure 2.13: Component selection dialog

Now you can stop the FileChooserDemo and restart it by executing the "Setup" node. The result of your efforts should look like this:

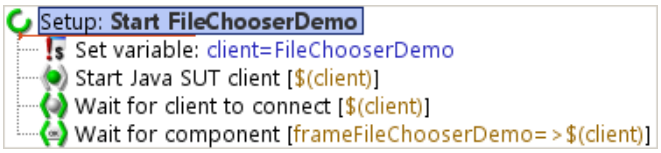
Setup: Start FileChooserDemo
Set variable: client=FileChooserDemo
Start Java SUT client [$(client)]
Wait for client to connect [$(client)]
Wait for component [frameFileChooserDemo=>$(client)]

Figure 2.14: Setup sequence of the FileChooserDemo

The next step is to make a test-case from the previously recorded clickstream sequence. For this we have to move the sequence from the "Extras" into the unnamed test-case. You'll need to open the "Test-case" node before moving the "Sequence" node inside. Let's rename the test-case to 'Clickstream' and the sequence to 'Some clicks'.

Lastly you can create a "Cleanup" sequence to stop the application. This sequence contains two nodes: One to stop the client and one to ensure that it actually terminated. From the figure below, you should be able to see what nodes are necessary to carry out this little exercise. Your test-suite should now look like this:

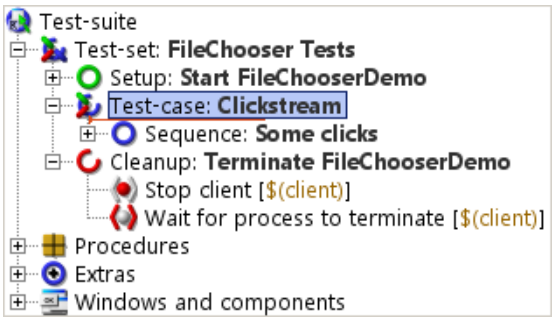

Figure 2.15: The Organization of your Test-suite

Now you've completed the most important steps of structuring a test-suite. In the following section you'll expand your test-suite by introducing a check of a specific text-field

2.5 Adding a Text-Check

To monitor the client's behavior we use check-nodes, which query certain states and properties of elements within the SUT. The first check that we'll introduce is a "Text-check," which verifies that a text-field contains the predicted string. Our check will be performed on a button within the FileChooser window.

If the SUT is not running, start it up now using your "Setup" sequence. Within the FileChooserDemo window, click the "Show FileChooser" button.

To begin recording of a check, click the "Record a check" button and switch to the FileChooser window. Now when you move your mouse over the components you will notice a blue border indicating the current selection. To record the check, move to the "Open" button, and right-click. You'll then see a popup menu appear which gives you the choice of possible standard checks for the button component, as seen below:

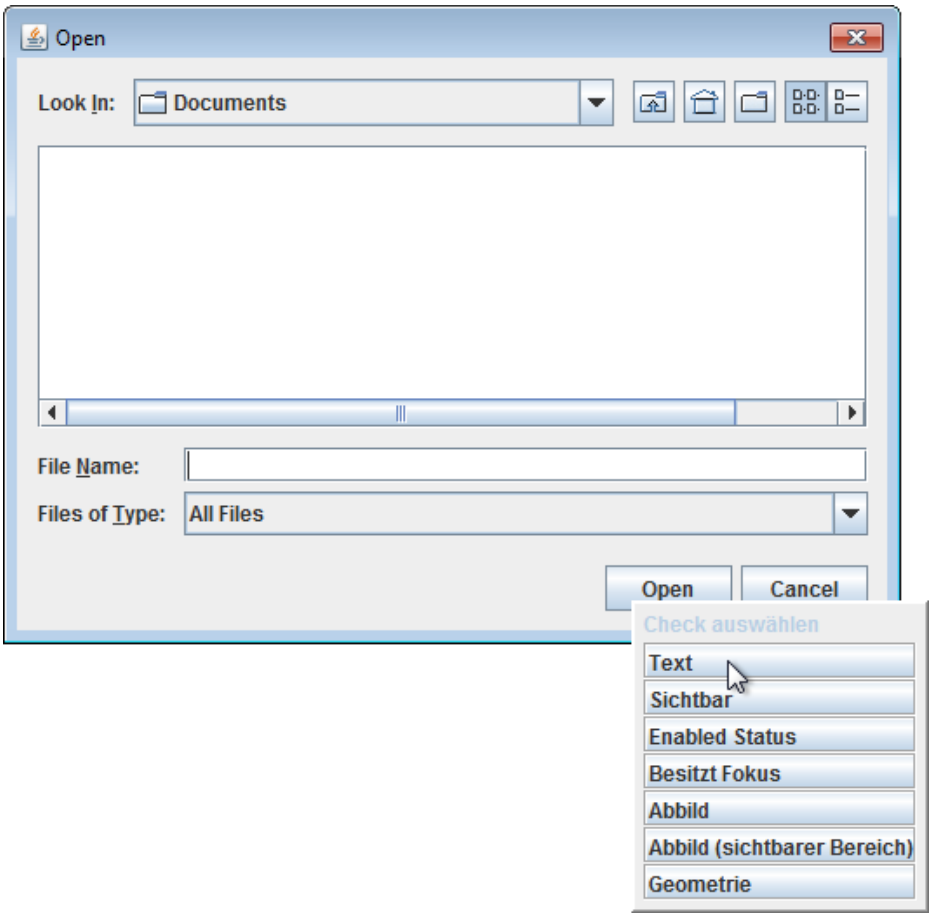

Figure 2.16: Recording a Check in the FileChooser Window

Choose ”Text” from the popup menu, return to your test-suite window and stop recording using the ■ button.

Note

Instead of returning to the test-suite to activate the check mode by pressing the ✓ button, you can also utilize F12 while remaining in the SUT. This hotkey activates/deactivates the record check mode.

Afterwards, record another arbitrary clickstream sequence using the ● button, and then go ahead and close the FileChooser window.

You should now have the know-how to organize the recorded sequences into a test-case. You can compare your results with the way we organized the suite and renamed the sequences. Additionally we transformed sequences into ”Test-step” nodes which specify them more clearly and make them also visible in the test report.

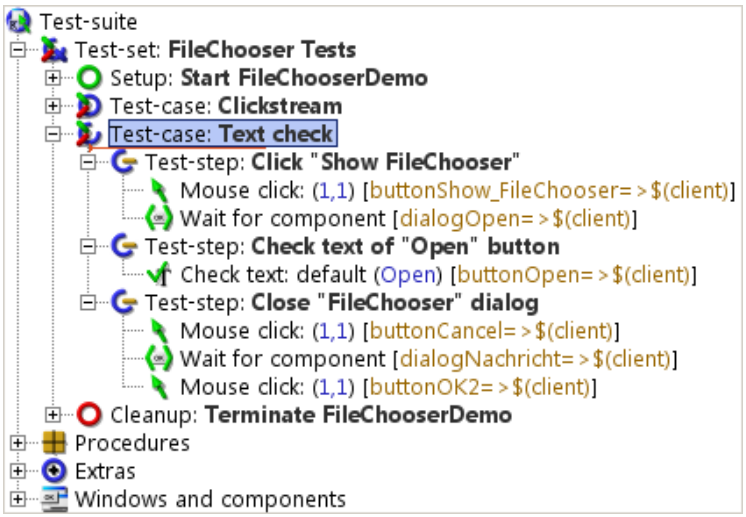

Figure 2.17: The Text-Check Test-case Organized into the Tree Structure

Note Transformation of nodes can be easily done via menu Operations | Transform node into or also a nodes context menu.

At this point, you can give your new test a shakedown run. Stop the SUT client now if it is running, click on the 'FileChooser Tests' test-set node, and start execution by hitting the replay button.

The result of the test-run is stored in the run-log. To view it just make use of the error dialogs "Open run-log" button or use the Run→1. ... menu option or take the short-cut Control-L.

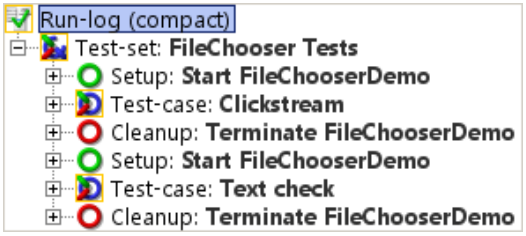

Figure 2.18: The Run-Log of Your Test-set

Within the run-log you see that the "Setup"/"Cleanup" pair is executed before and after each test-case.

Note Starting and stopping the SUT before and after each test-case is a safe way for a proper state but a time consuming one. Typically you will try to start the SUT only once and use others methods for resetting it to a defined status.

In this test there were no errors or exceptions in the test-run. But you can see yellow frames around some nodes, which indicate warnings. In case of the FileChooser they are caused by components which have no name assigned. We will not dicuss those warnings here any further, but you will find detailed information on the importance of component names in the user manual.

Upon closing the run-log window you will be prompted if you want to save the run-log (since it will only be available until QF-Test ends).

Let's modify the test now so that we cause an error in the text-check. Click on your "Check-text" node so that you see its details displayed as follows:

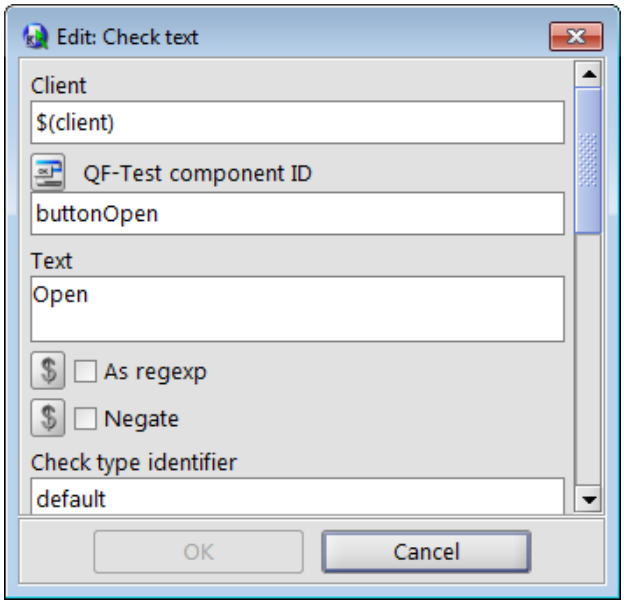

Figure 2.19: Properties of the "Check Text" Node

You see here that in the "Text" field a value has been entered ("Open") which represents the value that QF-Test expects. Change this text to some other arbitrary value and re-run the test.

Now when you replay the test, a dialog box appears at the end indicating that there were "0 Exceptions, 1 Error, 0 Warnings."

If you open the run-log, you'll see that red-fields now enclose certain nodes, indicating that some error occurred inside the child-nodes. You can use the run-log menu option Edit→Find next error to jump directly to the node which caused the error or use the Control-N keyboard shortcut.

We'll now go on by expanding your test-suite to include a test-case that checks the business logic of the SUT.

2.6 Checking Business Logic

Up to this point in the tutorial, we've primarily concentrated on the Swing implementation of windows and components within the SUT application. What we've not touched on is the idea of testing the actual business logic of the application, which will be our focus for this section.

To check the business logic we need a causal connection between some action of the user which leads to a reaction from the application. Within the FileChooserDemo, we've located one such feature which nicely demonstrates this type of connection in which a text field switches state from disabled to enabled upon clicking the "Custom" radio button (located in the left part of the window).

For our test, we've chosen to record a clickstream that activates the text field, then checks that the field really is enabled, and finally resets the application through a click on the "Open" button. This sequence is a true action-reaction test of the application since these two components are not connected in Swing. To just check the content of the text field is not a test of the logic but of the initial state of the whole application.

The steps for this test should be quite similar to those you did for the "Check text" test, with the clickstream sequence and the check recorded separately. We've provided our test-suite solution in the following image, to give you some assistance along the way:

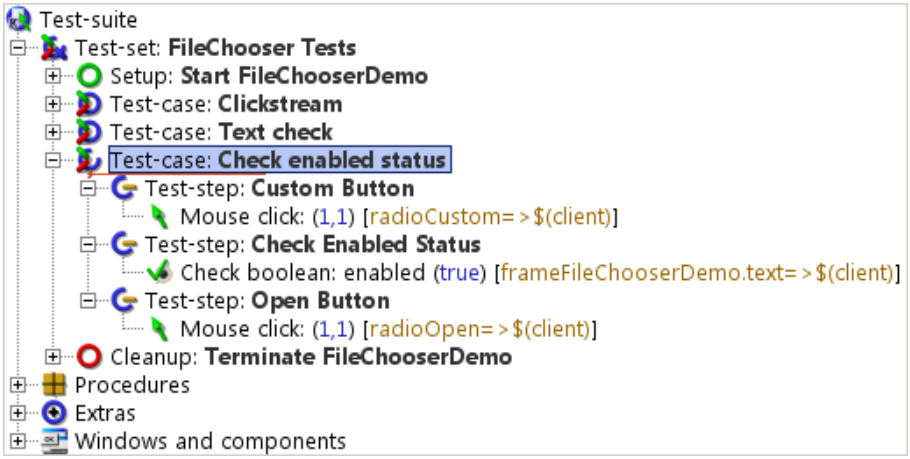

Figure 2.20: The Check Enabled Status Test-case

Here's a summary of what we did:

- The first sequence clicks the "Custom" button, thus activating the text field.
- The second checks that the text field is really active. Note here again that there is no direct connection in Swing between the button and the textfield, thus we are testing actual business logic of the application.

- The third sequence resets the application by pressing the ”Open” button.

With that we wrap up this chapter of the tutorial. In the next chapter, we’ll concentrate more on error diagnosis by diving into the QF-Test debugger.

Chapter 3

Using the Debugger [30-45 min]

3.1 Introduction

In this chapter we will learn how to run a test-suite with QF-Test's built-in and intuitive debugger. For those of you familiar with debugging programs in Java or other programming languages, you will find this debugger similar in function and usefulness.

For the debugger tutorial we will use a test-suite which you are most likely already familiar with, `Options.qft`. If this test-suite is not already running on your system, refer to section chapter 1(2) above to get it running.

3.2 Starting the Debugger

The QF-Test debugger can be started either by selecting a node (or nodes) to execute and pressing the step-in or step-over buttons, or by using the Debugger→Step In or Debugger→Step Over menu items. The sections below describe the function of these operations in more detail .

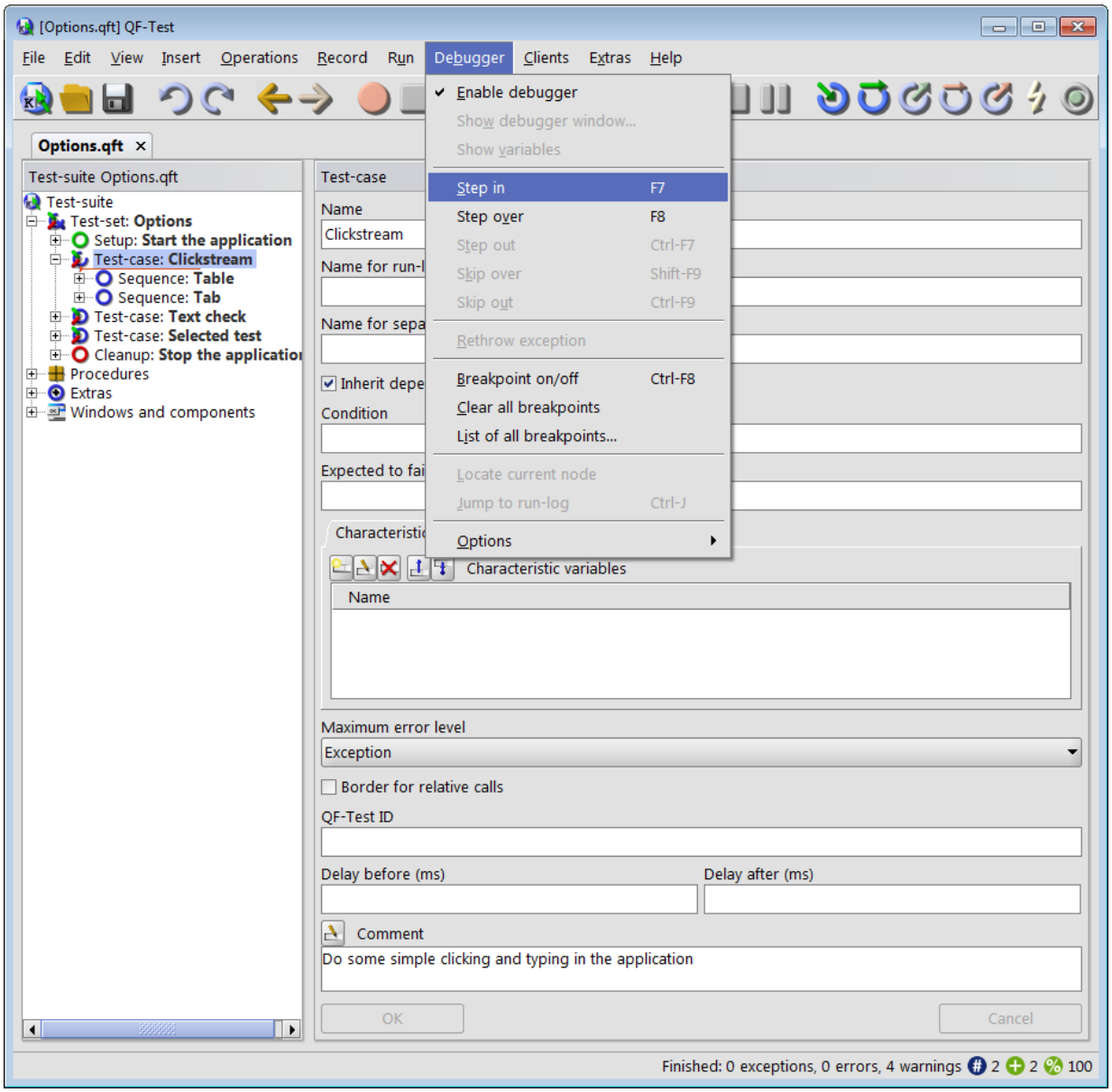

Figure 3.1: Select a node in the test-suite `Options.qft` and start the debugger.

Note that if you are running tests on your test-suite and use the play button to start execution, the debugger will normally not be seen. If you activate the debugger with the Debugger→Enable debugger menu item the debugger will be activated automatically if an uncaught exception is thrown. If a user-defined breakpoint is reached (q.v.) the debugger is activated unconditionally. In either case, execution of the test-suite will be stopped and the node which stopped execution will be indicated with an arrow.

3.3 The Debugger Window

The debugger is run either from within the normal test-suite view, or by opening the dedicated debugger window by selecting Debugger→Show Debugger Window once the debugger has been started.

You can also cause the debugger window to open automatically when the debugger starts by changing the global setting through the Debugger→Options→Always Show Debugger Window menu option.

The debugger window itself looks similar to the normal test-suite view which you are by now familiar with. It contains, however, only a subset of the QF-Test operations which are needed for debugging. The figure below shows the debugger window after starting the debugger for the `Options.qft` test-suite.

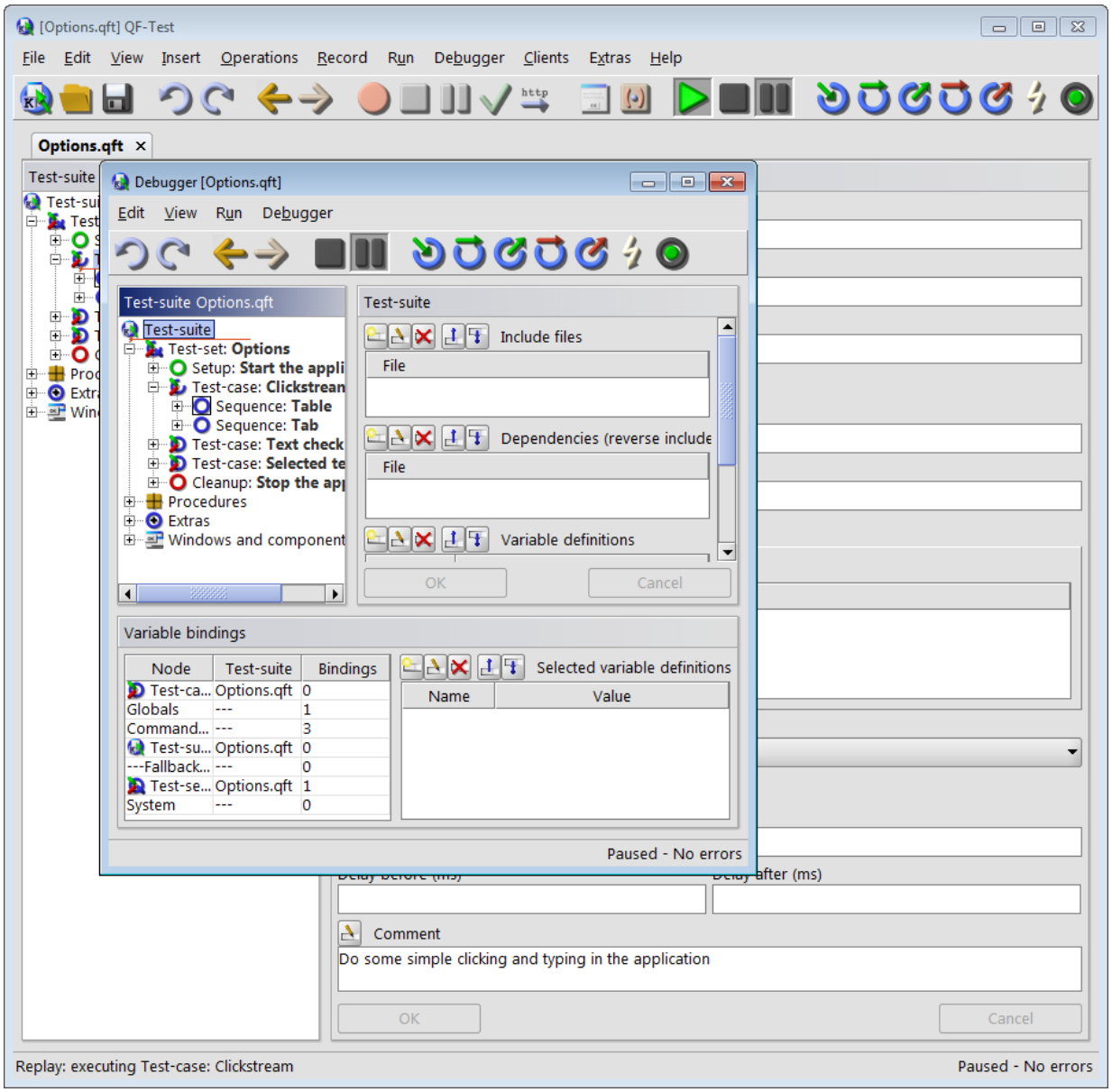

Figure 3.2: The debugger window.

3.4 The Current Node versus the Selected Node

When the debugger is running, an indicator is used to mark the ”current” node, which simply shows you where the debugger is while a test or sequence is being executed. Note that QF-Test marks the hierarchy of execution, from entry point down to the current node, with a small pointer as seen in figure 3.2[43].

If you cannot immediately see the current node at any time while the debugger is running, you can press the ”Locate Current Node” button or select the Debugger→Locate Current Node menu option to cause the debugger to ”select” the current node.

A ”selected” node is a node that is highlighted as when you click on it with your mouse. This can sometimes cause confusion, because the current node during debugging is often selected as well. Learn to recognize the distinction though!

3.5 Stepping Through a Test or Sequence

You are now ready to begin stepping through nodes of the test-suite to see the behavior of the debugger. With the current node set to ”Sequence: Table” as in figure 3.2(43), press the step-in button now.

You will see that the debugger causes the current node to expand to reveal the child nodes beneath it, and that the first child node is now selected as the current node, as shown in the figure below:

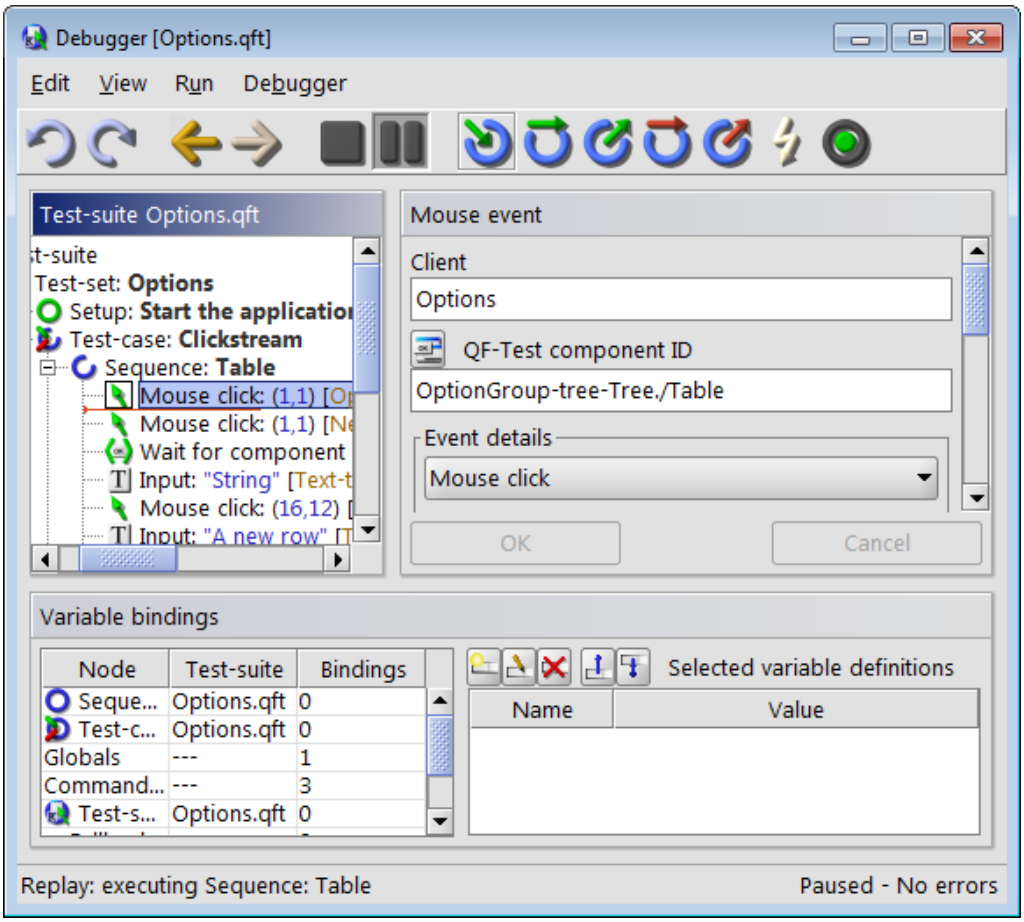

Figure 3.3: After a step-in operation.

Try pressing the step-in button a few more times. The debugger will execute each node, reset the current node to the next available node, and wait in a paused state for your next command.

Now after seeing the functionality of the step-in operation, press the ”Continue test-run” button to allow this particular sequence to run to completion.

After executing the first sequence of the test-suite, your debugger view should now show something similar to the figure below:

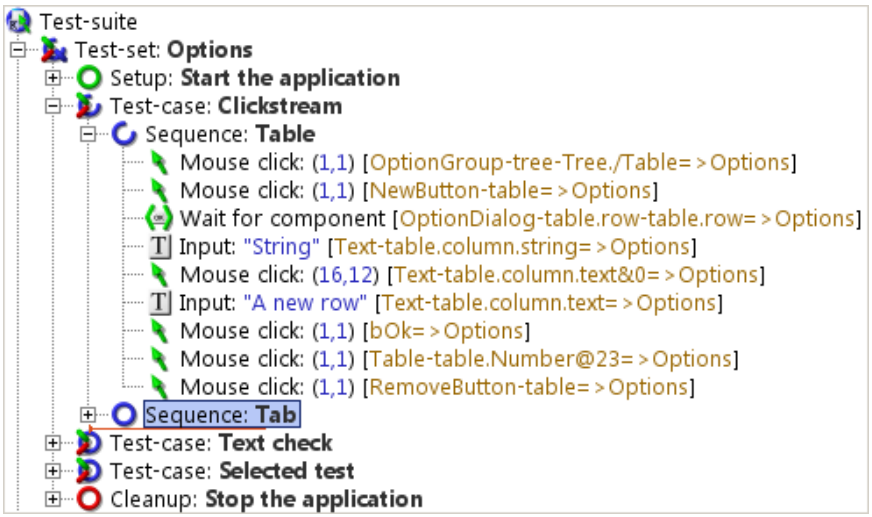

Figure 3.4: Preparing for the step-over command.

If this ”Sequence: Tab” node is not selected in your window as shown in the figure, then select it now. When you’re ready, press the step-over button.

This time, the debugger reacts by starting execution of sequence, but unlike the step-in operation, you will not see the individual steps. While the sequence is running, you can expand the ”Sequence: Tab” node to see that there several child nodes underneath it, indeed there are several other sequences located here.

The step-over operation is therefore a way for you to execute a node without having to concern yourself with the details of that node’s individual steps.

When execution of the ”Sequence: Tab” node has completed, the ”Sequence: Text Check” node will now be selected. Press the step-in button, so that your debugger view shows the following:

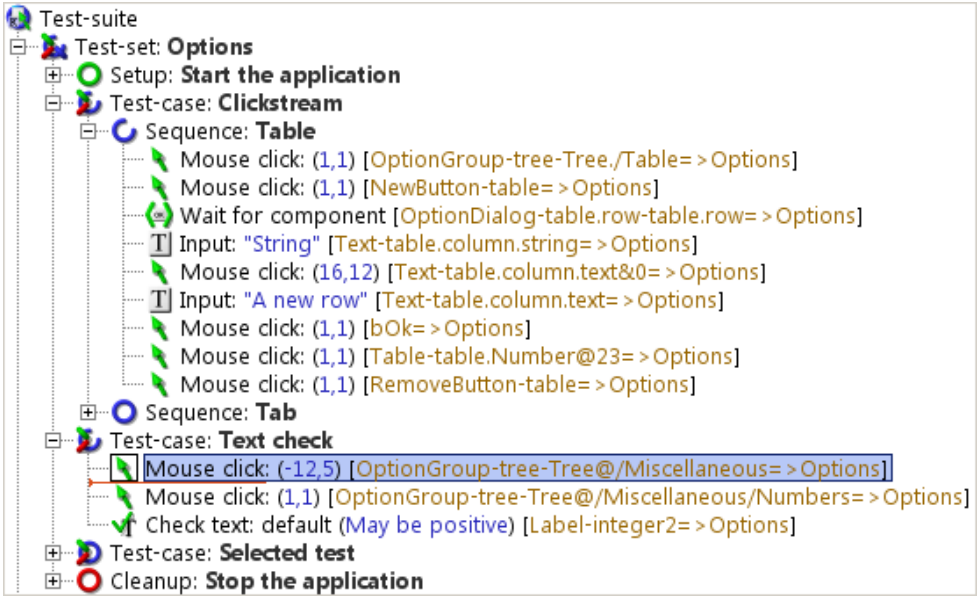

Figure 3.5: Preparing for the step-out command.

Now press the step-out button. You will see that the debugger causes the rest of the nodes in the sequence to be executed and the next ”Sequence: Selected test” node is now selected.

What does the step-out operation do? Very simply, it executes all nodes found at the same level in the hierarchy as the current node and stops when a node that is higher in the hierarchical structure is found. Any child nodes found during execution of ”step-out” are, of course, executed as would be expected.

From this example, the functionality of step-out may look similar to that of the ”Continue test-run” operation. This is partly true, but there is one distinct difference: The step-out operation stops at the next node that is higher in the hierarchy, whereas ”Continue test-run” works until it finds a node that is at the same level as the entry point node where you began. If this sounds complicated, simply try playing around with the two operations within the test-suite. You’ll quickly get the hang of it!

3.6 Skipping Execution of Nodes

The ”skip” functions expand the QF-Test debugger’s capabilities in a powerful way which is not typically possible for a debugger in a standard programming environment. In short, they allow you to jump over one or more nodes without having to execute them at all. This may be helpful due to different reasons, e.g. to navigate fastest to a certain position in your test run or skip a node which currently leads to an error.

Set up the test-suite again as you did in the beginning of the previous section, and press the step-in button again until the current node is inside of the sequence as seen in figure 3.3[45]. Now press the skip-out button. You see immediately that QF-Test simply jumped out of the sequence in which you were located without executing the remaining nodes in that sequence!

Your test-suite should now look like the image shown now in figure 3.4[46]. Leaving the set-up as is, now press the skip-over button. Here we see a similar behavior as with skip-out, QF-Test jumped over the selected node without executing any child nodes within it. (If you expand the node which you just skipped over, you see there are indeed several other sequences, none of which were executed).

One note about skip-over and skip-out: use them cautiously. Skipping out of a sequence before its completion could cause your SUT to be left in an unknown state that other sequences or tests in your test-suite cannot react to.

3.7 Setting Breakpoints

Setting a breakpoint is a deterministic way of allowing your test-suite to execute until it reaches the node you specify breakpoint, at which time the debugger will be active and allow you to continue as you wish.

To set a breakpoint, simply click on a node and select the Debugger→Breakpoint on/off menu item, or alternatively use the Control-F8 shortcut. The breakpoint is indicated with a (B) before the name of the node, as shown in the following figure:

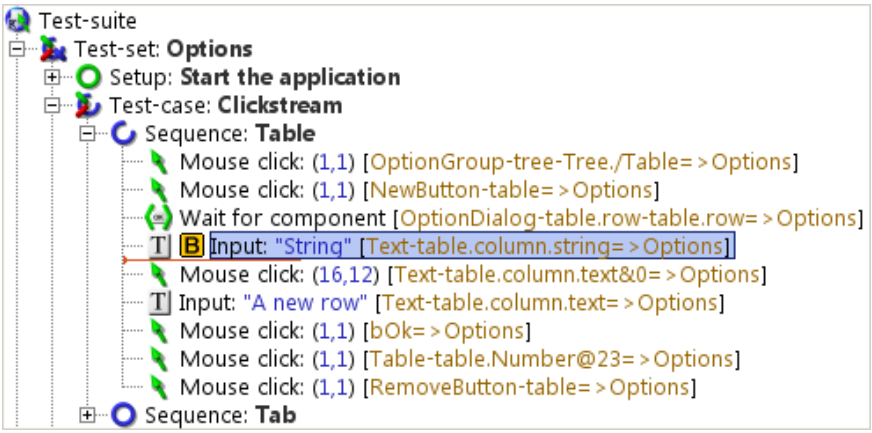

Figure 3.6: Setting a breakpoint.

To see for yourself how a breakpoint works, select the top-level node of your test suite and press the replay button.

To turn off a breakpoint, simply click on the node where the breakpoint was set, and select the Debugger→Breakpoint on/off menu option or Control-F8 shortcut again. The menu option Debugger→Clear all breakpoints is also useful to remove all breakpoints set in your test-suite.

There is no limit to the number of breakpoints you can set in your test-suite, but note that breakpoints are not saved with your session. So when you close the test-suite and open it again, the breakpoints will be gone.

3.8 Resolving Run-Time Problems

As you begin to create and run your own test-suites, you might face a problem like an unknown QF-Test component ID or procedure name. This is a common problem and is typically caused by an incorrectly entered QF-Test ID or name, or perhaps a change in the name or QF-Test ID which you are referencing. During run-time, you will see the manifestation of this problem as a halt in the execution of your test-suite, with an appropriate message from QF-Test relating to the unknown element.

This gives us a chance to demonstrate one of QF-Test's most powerful features in the debugger: the ability to correct such problems on the fly during run-time execution of your test-suite.

Let's first cause your test-suite to throw an exception when it runs into an unknown QF-Test component ID. Take any node within the test-suite and edit the QF-Test component ID field so that it is obviously wrong. For this example, we took the first node under the "Table" sequence we used in figure 3.6(48)

Figure 3.7: Select a node to modify.

and modified the QF-Test component ID entry to look like this:

Figure 3.8: The modified node.

To edit the QF-Test component ID field, click on the node and then select the Edit→Edit... menu option, or simply edit the field in the Details window (View→Show Details if the Details window is not visible on your screen). Click OK in the Edit dialog or the Details window after making your change. At this point you will see a warning that the QF-Test component ID you entered is not valid. That's good! This is QF-Test's way of attempting to find problems before they occur at run-time. However, for this example we want to force an incorrect reference, so just leave the modified entry as it is.

We can now let the modified test-suite run. Select the "Test: Clickstream" node and press the play button to start execution. The test run will quickly stop with an error message dialog box:

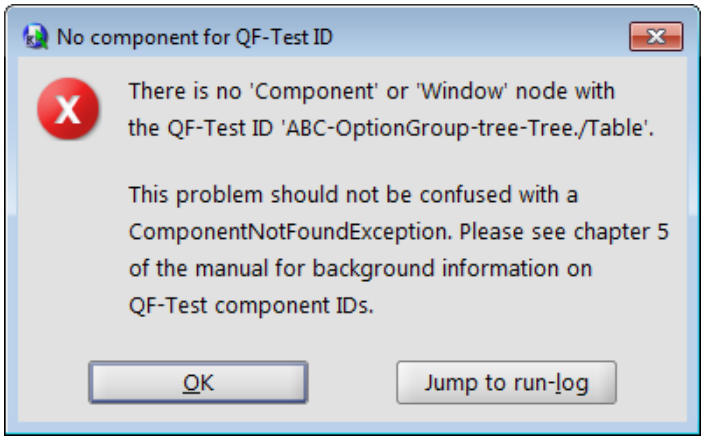
No component for QF-Test ID
There is no 'Component' or 'Window' node with the QF-Test ID 'ABC-OptionGroup-tree-Tree./Table'.
This problem should not be confused with a ComponentNotFoundException. Please see chapter 5 of the manual for background information on QF-Test component IDs.
OK
Jump to run-log

Figure 3.9: No component for QF-Test ID.

Click OK to close the dialog box. You will now see that execution of your test-suite is paused with the modified node selected as the current node. The debugger at this point is simply waiting for your directions. Do you want to skip over the offending node, completely halt the execution of your test-suite to investigate the problem further, or modify the node on the fly and continue execution? In this case, we want the latter option.

So with the debugger still paused, you can now edit the QF-Test component ID field again to change it back to its original state. If you've forgotten what the original QF-Test ID was, select the Edit→Undo menu option to restore it (assuming you've not made any other changes to the test-suite).

With the correct QF-Test ID restored, press the "Continue test-run" button to allow execution to continue where it left off. You'll now see that the test-suite runs just like before!

3.9 Jump to Run Log

At times, tracking down errors in your test-suite is not so simple as in the example we just showed. The actual cause for some problems may lie farther back in a sequence of events which led to the eventual exception or error which you see in front of you.

Under such circumstances, the run-log can be of immense use for debugging purposes, as it shows in detail the steps QF-Test used to execute each node and how it resolved QF-Test component IDs, procedure names, variables or other elements.

Since the run-log can be tedious to search, especially if your test-suite became quite complex, the debugger provides you with a simple mechanism to jump immediately to the region of interest in the run-log when an error occurs.

We'll demonstrate this by using the same simple example we used in section 3.8(49), which caused an exception to be thrown due to an unknown QF-Test component ID. Repeat the steps from that section and execute the test-suite as before, except that now instead of correcting the QF-Test component ID as we did before when the error occurred, we will first inspect the run log. To jump to the appropriate place in the run log, select the Debugger→Jump to Run-Log menu option, or use the Control-J shortcut. The run-log will now be opened, and the offending node will be automatically selected as shown below:

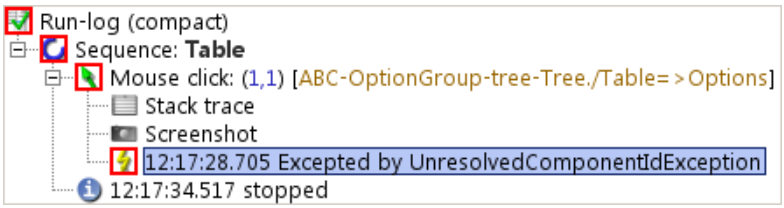

Figure 3.10: Jump to run-log.

Take some time now to look at the various elements presented for you in the run-log. Although the example we are using now is quite readily understandable, becoming familiar with this powerful tool will help you greatly when your test-suites become more complicated.

When you're finished, you can continue with the example as before or simply stop the test run. However, remember not to save the changes to the test-suite if you did not correct the error.

Chapter 4

Writing a Procedure [30-45 min]

In this and the following chapter we begin to explore some of the more advanced concepts of QF-Test by using a hands-on approach that will give you a good overview. We'll introduce (among other things) the important and useful concept of procedures within QF-Test. Along the way you'll pick up other insights that should prove helpful for developing your own test-suites.

As a start you'll create a simple procedure that will select a checkbox within the SUT client.

4.1 Getting Started with a New Suite

At this stage in the tutorial, you are ready to create your own test-suite which you will use to build upon in successive steps. From the QF-Test menu, select the File→New Suite option.

Note

As you've indubitably seen throughout this tutorial, shortcut keys are available for most menu options. The shortcut keys are a quick way to perform useful operations with a simple tap on the keyboard. The File→New Suite menu option, for example, can also be done with the Control-N shortcut. You'll see shortcuts listed beside most menu options in QF-Test, but many other shortcuts are available that are not immediately obvious from the GUI. For a complete list, refer to the Keyboard shortcuts section of the user manual. You can also find a little helper there for attaching to your keyboard which shows the function key assignment of QF-Test.

A test-suite, of course, needs an SUT client to test with. The Options demo, which you are most likely familiar with from earlier chapters of the tutorial, is a very simple application ideally suited for our needs as an SUT. Our next step is to establish this application as an SUT client for your new test-suite.

To get the Options demo setup as the SUT for the test-suite, you'll need to create a Setup node which will launch the application and establish the link between the application and QF-Test. For this step, you can simply copy the logic from `Options.qft` and use it in your new test-suite. To do this, open up `Options.qft` (refer to chapter 1[2] if you're not familiar with this test-suite) and expand the "Test: Options" node. The first visible node will be the "Setup: Start the application" node. Click on this node and copy it with Edit→Copy.

Now return to your new test-suite. You'll need to expand the "Test" node so that an insertion is possible, then you can paste the copied node in with Edit→Paste.

A Cleanup node is usually associated with a Setup node and is used to undo the steps performed by the Setup. In this case, we want the Cleanup node that will stop the SUT application. From the `Options.qft` suite, copy the "Cleanup: Stop the application" node and paste it into your new test-suite right after your Setup node.

There's one other step you need to do before this section is complete: copy the windows and components from `Options.qft` to your new suite. The reason for this step may not be immediately apparent to you at this stage, but we will get into a more detailed discussion of components in a later section. For now it suffices to say that the component descriptions stored under the "Windows and components" section of a test-suite are the way that QF-Test recognizes elements such as menus and buttons in an SUT. Simply expand the "Windows and components" node of `Options.qft`, select the nodes you see there and then use the Copy and Paste functions to copy them into the same node of your new test-suite. If you get confused, you can refer to the diagram below to see how things should look.

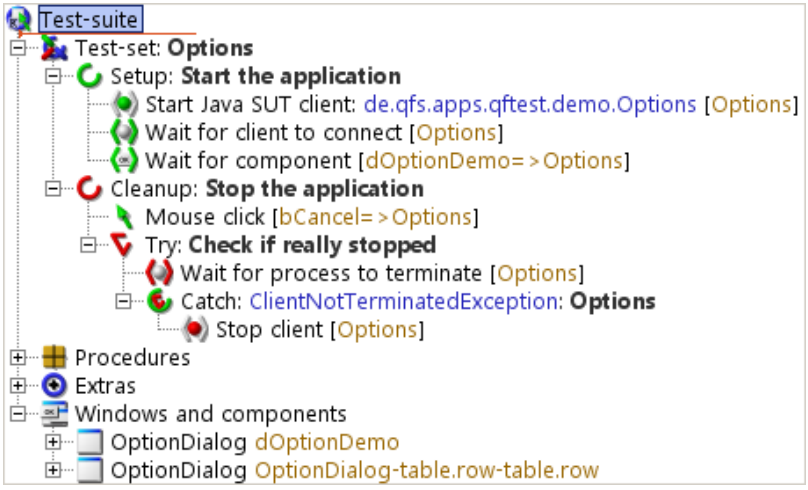

Figure 4.1: Skeleton for new Test-suite

At this point, you've created a working skeleton for your test-suite that we'll populate in

later sections with various procedures and tests.

For now, you can get the SUT running by selecting the the Setup node and clicking the replay button. Once the SUT is up and running, bring up its window and select the ”Miscellaneous” node in the ”Preferences” tree. This will bring up several miscellaneous options for the demo, among which is a checkbox labeled simply ”A BooleanOption” as such:

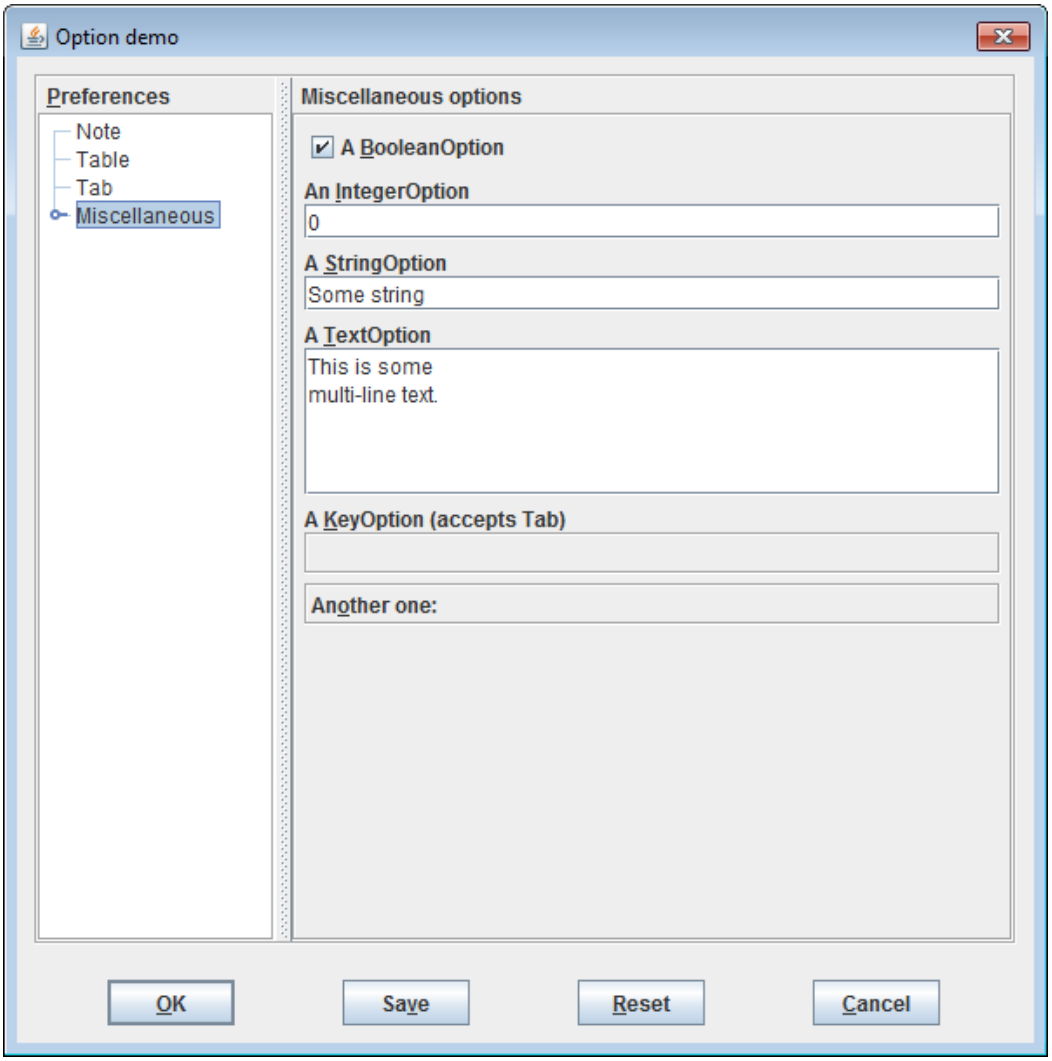

Figure 4.2: Checkbox within the Miscellaneous Options of the SUT

This is the checkbox that you will be manipulating with your new procedure in the next few sections.

4.2 Create the Procedure

Being a programming construct, a procedure is typically defined as a method used to accomplish a repetitive task - such as opening a window or checking if a certain key was pressed. For this part of the tutorial you will be creating a procedure that selects a checkbox within the SUT.

First, expand the Procedures node within your new test-suite.

Note QF-Test gives you good visual clues as to whether a node is collapsed or expanded. For example, the Procedures icon changes from (collapsed) to (expanded).

Now click on the Procedures node so that is selected, and select the Insert→Procedure nodes→Procedure menu option.

A new dialog window should appear, allowing you to enter the description of the new procedure. Give it the name ”selectCheckbox” and hit the OK button. Here’s how the test-suite should look after adding your first procedure:

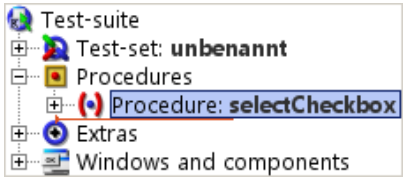

Figure 4.3: Adding a Procedure

Note Properties of a node within QF-Test (such as the name of the procedure) can be changed simply by clicking on the node and then editing these properties in the details window, or by right-clicking on the node and selecting Edit... . In either case, you’ll need to hit the OK button to confirm any changes you’ve made.

4.3 Add a Check Node

We’re now ready to start adding some real substance to the procedure. But first consider how you might proceed in setting the checkbox. One of the first steps we’ll need to perform is checking the state of the checkbox before we attempt to set it. In case it is already checked, there is nothing to do but leave it this way. Otherwise (it is unchecked), we can go ahead and set it.

Such checks are done in QF-Test with so-called ”check-nodes,” which allow you to perform various types of queries about the state of elements within the SUT. The one we’ll start with is called the ”Check boolean: selected” check-node. As its name implies, it simply queries the state of a component to see if it is selected or not.

You've hopefully already had some experience with check-nodes from the previous chapter in this tutorial. If not, it might be helpful to refer back to section 2.5(34) now. To add the "Check boolean: selected" node, you will now perform similar steps.

First bring up the SUT window and select the "Miscellaneous" node in the "Preferences" tree as we showed in figure 4.2(54). This will allow you to easily get to the correct component once you start recording. Now return to your test-suite and click the record-check button and then return to the window of the SUT. As before, when you move the mouse over the various components, they invert their color, which indicates selection. Select the "BooleanOption" checkbox which we identified earlier and right-click it. The pop-up menu which appears contains, in addition to the various types of checks we saw in the previous example, an option for "selected state," as shown in the figure below.

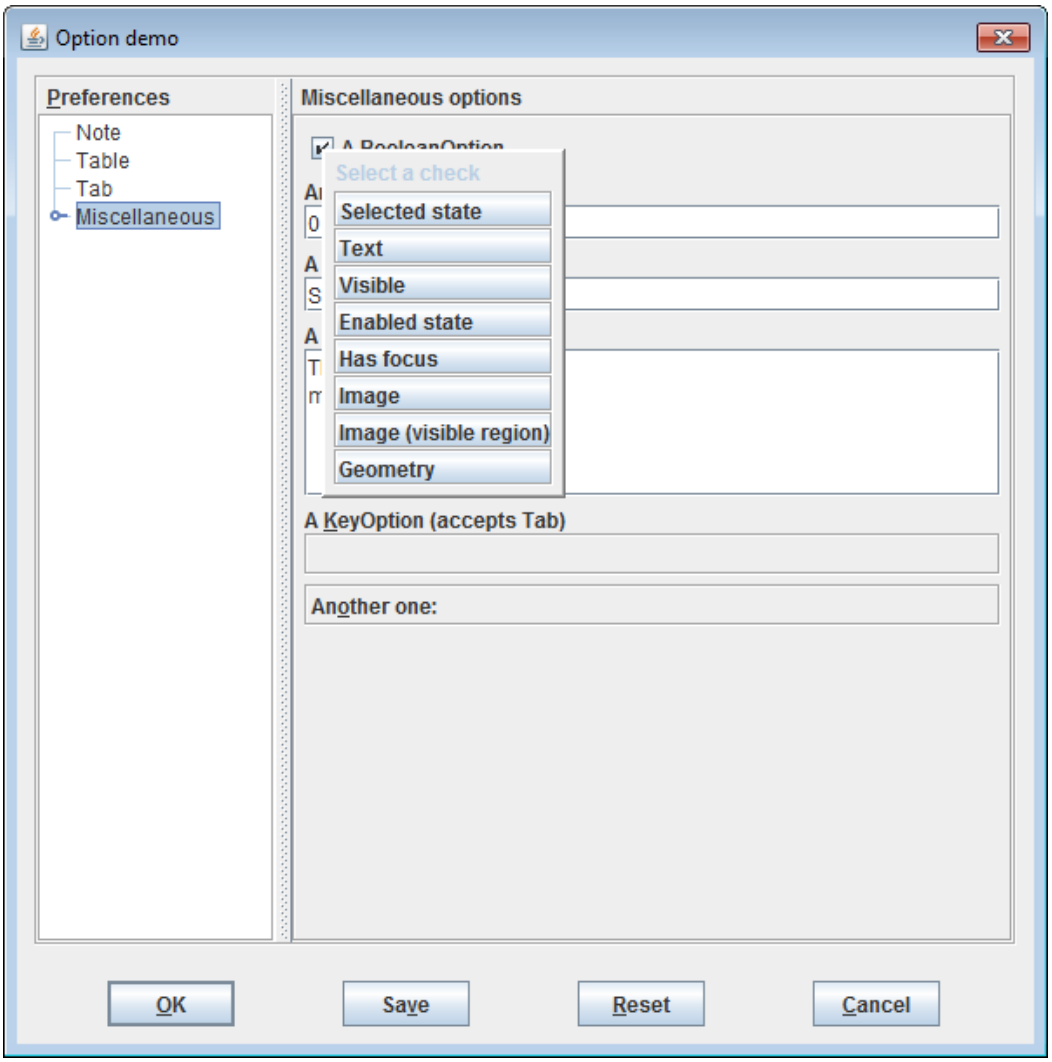

Figure 4.4: Recording the "Check boolean: selected" Node

Choose the ”selected state” option, return to your test-suite and stop recording with the stop button.

QF-Test will place the recorded sequence within the ”Extras” node of your test-suite, which will be immediately visible after you stop recording. If you expand the recorded sequence, you’ll see that our ”Check boolean: selected” node is there. This node must now be transferred from its current location to your procedure. You can transfer either just the node itself, or the entire recorded sequence. Since there is only one child-node inside sequence, the difference is purely aesthetic.

Do this using the Copy or Cut function and then the Paste function to place it inside your procedure. Make sure to expand the ”selectCheckbox” procedure node before pasting the copied node, otherwise it will not be properly placed within the procedure, as shown in the figure below.

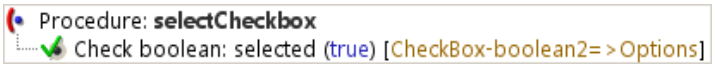

Figure 4.5: A ’Check boolean: selected’ Node within the Procedure

Note

QF-Test provides a feature which allows you to directly insert a recorded sequence at a selected location, thus saving the work with cutting and pasting. Select the Edit→Options menu option. From within this window select the ”Record” node to bring up the recording options, then tick the checkbox ”Insert recording at current selection.”

Now give it a try! Select your procedure and click the replay button. Not much happens at this point, but the lower status bar indicates the procedure completed successfully. The first step is complete.

4.4 About Components

Before we continue, it might be helpful to give you some brief background information about checks and components which may not be immediately evident from the exposure you’ve had to them in the tutorial thus far.

A component is simply an element of the SUT, such as a button, a menu, or (as in our case) a checkbox. Each component should have a unique QF-Test ID, which is registered within your test-suite, so that QF-Test can find the component when executing the test-suite.

If you click on the ”Check boolean: selected” node which you recorded in the previous section, you will see its properties appear in the details window on the right. Among these properties is a field called QF-Test component ID which should be filled in with the

identity of the checkbox we identified in figure 4.2(54). You'll see that this component has been given the QF-Test ID of "CheckBox-boolean2".

This QF-Test ID is in and of itself just a name, a handy way for you and QF-Test to reference the same component. These QF-Test IDs are defined within the "Windows and components" section of your test-suite whenever you record something from the SUT. If you expand all the nodes within this section, as shown in the figure below, you'll eventually find the component we recorded for "Check boolean: selected".

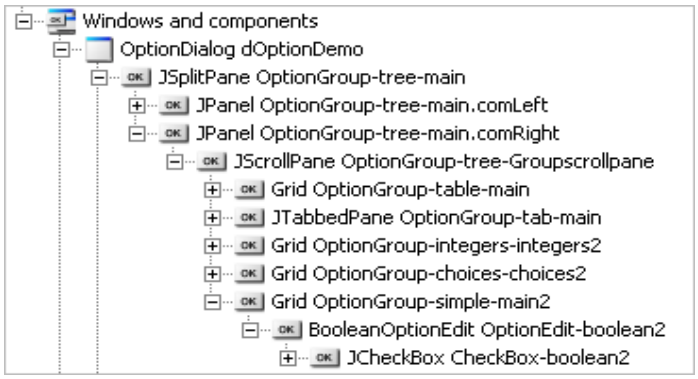

Figure 4.6: The Checkbox omponent within the component tree

Note

An easy way to locate a component is to use QF-Test's locate-component feature. Bring up the popup menu by right-clicking the "Check boolean: selected" node and select "Locate component" or simply select the node and press Control-W. This will take you right to the Component node for the check. Pressing Control-Backspace will take you back to the check node.

Go ahead an click on the node with the "CheckBox-boolean2" QF-Test ID. You'll see a bit about how QF-Test internally identifies components.

Component recognition within QF-Test is a complex subject, and we will not dwell on it further than the overview just given. For more details about components, please refer to the QF-Test user manual.

4.5 The Try/Catch Mechanism

But what happens if our checkbox component is not checked? Uncheck the checkbox now and replay the procedure. You should see a message like this come up when the procedure completes:

Figure 4.7: 'Check boolean: selected' Error

The error occurred because your "Check boolean: selected" node was expecting the checkbox to be checked.

Click on the your "Check boolean: selected" node again so that you can view its properties in the details window. Within these details, you'll see fields marked "error level of message" and a checkbox to "Throw exception on failure". Using these properties, you can modify the action that QF-Test will take when a check fails. Let's change these settings and observe the results after replaying your procedure.

Ultimately, we want this check to throw an exception when it fails. This will provide an opportunity introduce the very useful try/catch mechanism. Set the "Throw exception on failure" checkbox in the properties as seen below:

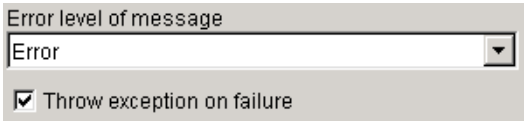

Figure 4.8: Throw Exception on Failure

The try/catch mechanism is a common technique used by most programming languages to handle exceptions (i.e. unexpected conditions) when they occur. The concept is simple: 'try' to perform some action; if an exception occurs during the action, then 'catch' it and perform some other sequence of operations. If you're not familiar with this construct, don't worry - it will be come clear enough once you see it in action.

Insert a "Try" node right at the beginning of your procedure. To do this, click on the procedure node (the insertion point always is right after the node that is selected), then select the Insert→Control structures→Try menu option. A dialog will appear for you to enter the properties of the node. For this node, you're not required to fill in any of the fields, however, for readability, it is sometimes helpful to enter a name. We wrote "checkbox selected".

Expand the "Try" node so that you can insert nodes inside of it. All actions taken within this "Try" block can potentially be caught by a "Catch" node if an exception arises (which

will be explained shortly). Thus, we now want to insert the action that we want to "Try" - which is our "Check boolean: selected" node. This node should still be in your procedure, but probably not in the right spot. Use the Cut and Paste functions to insert this check-node immediately after your "Try" node.

A "Try" node is useless without "Catch", so this is the next step. Insert a "Catch" node after your check-node (but still within the overall "Try" block) with Insert→Control structures→Catch. Again, a dialog window will appear for you to fill in the properties of the node. For the "Catch" node, you do need to define what type of exception should be caught. The first field is labeled "exception class," which contains a list of all possible exceptions. Select "CheckFailedException".

Figure 4.9: Selecting an Exception to Catch

The try/catch mechanism is meant to be a deterministic construct. That is, the exception (or exceptions) to be caught relate directly to the actions undertaken in the "Try" block. In our case, our "Try" block will contain a check-node to "Check boolean: selected," so the exception to be caught is the "CheckFailedException". This, of course, may not be immediately evident to you when implementing the try/catch mechanism. If you're not certain about what kind of exception it is that is being thrown that you need to catch, the run-log can prove useful. Try replaying your procedure again. Don't worry that the try/catch nodes are not complete. Of importance is that the "Check boolean: selected" node is still present and has its "Throw exception on failure" property set. Afterwards, open the Run menu and select the most recent run log (the most recent is always in position "1"). Scan through the run log and expand the nodes. You should then eventually see the logged item containing the thrown exception, as follows:

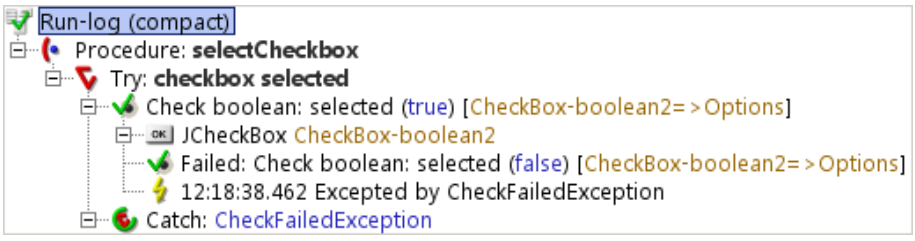

Figure 4.10: CheckFailedException in the Run-Log

Note For more information about exceptions and a complete list of possible exceptions, see

the user manual.

At this stage your procedure should now look something like this:

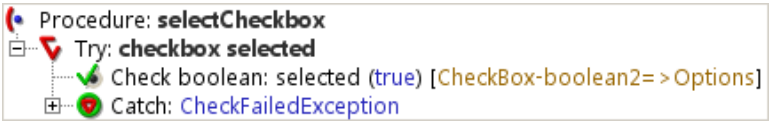

Figure 4.11: Procedure with Try/Catch

Note Let QF-Test help you out if you have questions about a certain node: just right-click on a node you would like help with and pick the ”What’s this?” option from the pop-up menu - QF-Test will take you right to its description in the user manual.

4.6 Complete the Selection Logic

Your try/catch nodes are in place, the only step left is to complete the actions that will take place if the ”CheckFailedException” is thrown and caught. Recall that the exception will be thrown if the check fails, i.e., if the checkbox is not set. Thus the action we will want to take if the exception is caught is to set the checkbox.

Here’s where we get to record another sequence from the SUT. Click the ● button and bring up the SUT window. The only thing we want to record is a click on the ”BooleanOption” checkbox. So click on the checkbox now and change the state from set to unset, or vice-versa (it doesn’t matter, we just want to record the state change). Now return to your test-suite and click the ■ button. After stopping recording, QF-Test will place the recorded sequence inside the ”Extras” section of your test-suite. If you take a look now inside the recorded sequence, you’ll see a ”Mouse click” node. If you take a look at the node’s properties, you’ll see several familiar items such as the QF-Test component ID of the checkbox.

Note You’re not required to make changes to the properties of this node, but adding a comment within it can sometime be helpful. Comments have no affect on the test, they aid only in the readability of the test-suite.

As we did previously when using a recorded node, you’ll need to move this ”Mouse click” node from ”Extras” to your procedure using the Copy/Cut and Paste functions. Expand the ”Catch” node within your procedure and insert the new node after it. Your procedure should now look as follows:

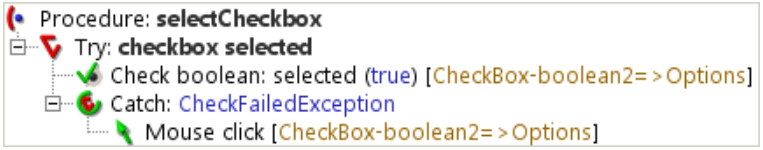

Figure 4.12: Procedure with Completed Try/Catch Mechanism

Your procedure is now a complete, functional unit. Click on the procedure node and try running it a few times using the replay button, in between changing the state of the checkbox within the SUT so that you can see the two different logic paths being executed. Using the Step-In function of the debugger (see chapter 3(40)) to step through your procedure is an excellent way to see exactly how the try/catch mechanism is functioning.

Just a small remark: A library which contains a set of such useful utility procedures like ”selectCheckbox” is part of your QF-Test distribution. This standard library is discussed in detail in chapter 7(83). However, we would like to spend a bit more time on improving our procedure first.

4.7 Making the Procedure More Usable

We’re going to make one last improvement to the procedure before wrapping up this chapter of the tutorial, something which may have already occurred to you. The problem with this procedure is that it works for only one specific checkbox. What if the SUT had more than one checkbox that we want to set? Would we write separate procedures for each individual checkbox?

The solution comes to us by way of variables. A variable can be used to substitute the QF-Test component ID attribute of the checkbox so that the procedure can be used for any checkbox within the SUT.

Variable syntax in QF-Test, as with any computer program, is quite specific. A variable is defined simply with a name. However, when referring to a variable, the required syntax is: `$(name)` - you’ll learn more about this in the following sections.

Let’s replace references in the procedure to the specific checkbox with a variable now. Call the variable `$(id)` for example. Note that there are two references to the checkbox component, one in the ”Check boolean: selected” node, the other in the ”Mouse click” node. To make the replacement, simply bring up the node’s properties and locate the QF-Test component ID field. Replace the text `CheckBox-boolean2` with `$(id)`. The procedure should now look as follows:

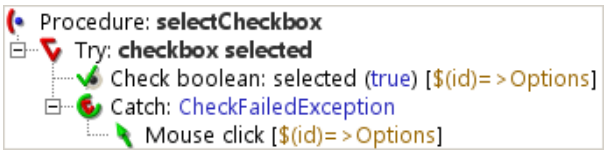

Figure 4.13: Variable Replacing Fixed Component

4.8 Calling the Procedure

If you attempt to replay the procedure now, you'll get an error indicating that "The variable 'id' was not bound". This is QF-Test's way of letting you know that there is no value set for the variable.

So how do you set a value for the variable? For the example we are using, the most common and useful way is to 'call' the procedure with a specific value for the variable. In technical terms this is referred to as 'passing an argument (or parameter)' to a procedure.

Let's insert a call to the procedure now. We'll return to the main "Test-set" node of your test-suite. Collapse the "Setup" node so that you can insert a node after it. Then we first need a "Test-case" node by Insert→Test and sequence nodes→Test-case for which you can assign an arbitrary name. After expanding the "Test-case" we can create the procedure call either by Insert→Procedure nodes→Procedure call menu option or by pressing Ctrl-A. As usual, a dialog window will appear for you to fill in properties of the node. For this procedure-call node, you'll need to fill in a few things. First enter the name of the procedure in the first filled, which is the "selectCheckbox" procedure.

Note

The third and possibly simplest way is to use drag and drop. When dragging the "selectCheckbox" procedure node from the procedures up into the "Test-case", the node is automatically converted into a "Procedure call" (as "Procedure" nodes are not allowed here). You will get the same effect when copying the procedure node e.g. by Ctrl-C and paste it into the "Test-case" by Ctrl-V.

Next we want to add the variable that will be passed as an argument to the procedure. This is accomplished with the "Variables" table that appears after the procedure name in the properties window. Click on the first icon for this table to add a new row. You'll see a new dialog window appear called "Edit table row" which allows you to make a new entry. Within this window, enter `id` as name and `CheckBox-boolean2` as value. When you're done, the completed properties for the procedure call should look like this:

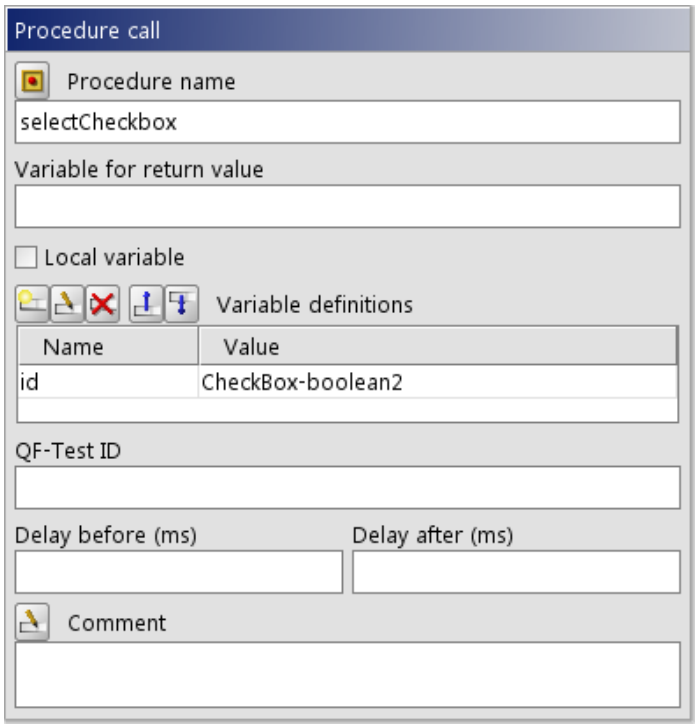

Figure 4.14: Properties of a Procedure Call

Note

Aside from the icons which serve as functions for adding, editing and removing elements in the variable list for the procedure call, you'll notice another icon in the above image: the small yellow and red button located after the "Procedure name" field. This icon is used to bring up a very helpful list of procedures located within your test-suite, in case you cannot immediately recall what the procedure is named. In the last that appears, locate the the "selectCheckbox" procedure and hit OK. QF-Test will automatically fill in the name for you in the properties.

Once the properties for the procedure call are completed, click OK to complete the node. You can now select the new node and click the replay button. You'll see that the call now passes the QF-Test ID of our checkbox to the procedure so that "selectCheckbox" can work properly. If the SUT had multiple checkboxes, you could now add several calls to "selectCheckbox," each with a different QF-Test component ID as the argument to the procedure!

In case you attempt to run the complete test-suite you will be prompted with an error message (Components not found) indicating that the target component `Checkbox-boolean2` could not be determined. This issue will be handled in Chapter 7.3 so you could skip this section, but we can also tackle the problem here for practicing purpose and better understanding. You might want to have a look into the run-log, especially the included screenshot will prove helpful.

As you can see, the problem happens due to the fact that there is no such component on the start window of the Option demo. First of all we need to record a sequence that expands the ”Miscellaneous” item. You should be able to manage that easily (open the SUT, click the ”Start recording” button, click on the ”Miscellaneous” node and click ”Stop recording”). Now where do we insert this sequence in our test-suite? Intuitively you might be inclined to place it above the procedure calls, but then a test run would come up with the very same error message. A look at the run log shows you why - because each sequence between ”Setup: Start the application” and ”Cleanup: Stop the application” will include a new start AND a stop of the SUT.

So in order to achieve that desired state of the SUT for all test cases, you are bound to include this sequence in the Setup itself. Alternatively you could create a new sequence to include the recorded mouse-click and the procedure call (in case only this procedure is meant to start in the given state).

Chapter 5

Creating a Generalized Procedure [30-45 min]

In this chapter we'll show you how you can make the procedure you wrote in the last chapter even more useful. With the introduction of packages, you'll see how you can build your own utility library to serve general-purpose functions.

5.1 Creating a Package

As you begin to write more procedures, you'll soon find that - like important documents strewn haphazardly about your table - things become cluttered and disorderly. Like a file-folder that helps you organize your documents into different categories, a package in QF-Test helps you group together your procedures. Our definition of a package is thus simply a grouping of procedures and/or other packages. We tend to think of packages as containing similar elements. However, you can organize your packages however you see fit.

In the last chapter you wrote a checkbox utility procedure called "selectCheckbox," which allowed you to check a checkbox in the SUT. We could, however, envision other such utilities for a checkbox. What if you want to uncheck a checkbox or query whether a checkbox is greyed-out/disabled? There are any number of possibilities, fitting one general category, i.e., checkbox utilities. So let's create a package for these procedures.

Click on the "Procedures" node of your test-suite and select the Insert→Procedure nodes→Package menu option. This will bring up the properties dialog window in which you have to insert only a name for the package. Call it "checkbox".

5.2 Move the Procedure

You already created one procedure which deals with a checkbox which can now be moved into your new package. Directly use the mouse or the Cut and Paste functions to move the ”selectCheckbox” procedure into the ”checkbox” package. A dialog will pop up and ask whether to update all references to this node. Please select ”Yes” and let ”Show updated references afterwards” activated, which will give an overview of all adapted nodes. In our case it is just one procedure call.

You’ll note that the name of your procedure is somewhat redundant now that it is inside the ”checkbox” package. Let’s change the name of the procedure from ”selectCheckbox” to just ”select”. This is but one small advantage of package organization: anything inside of the ”checkbox” package can be assumed to deal with a checkbox. Again you will be asked whether references are to be adjusted, which of course is helpful.

Here’s how your test-suite should look like now:

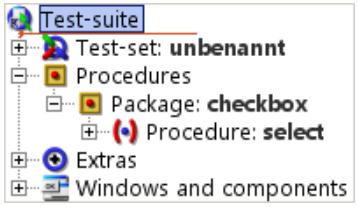

Figure 5.1: The Checkbox Package

5.3 Create the Deselect Procedure

Our next step will be to expand the package a bit by adding another utility procedure. A sensible choice would be, of course, a ”deselect” procedure which will uncheck a checkbox.

Your new ”deselect” procedure will work essentially the same as ”select” with only some minor changes in the logic. Let’s start by just making a copy of ”select” and pasting it into the ”checkbox” package. Don’t be concerned that you’ll have two procedures with the same name in the package, you’re going to fix that next.

You should now have two identical procedures in the ”checkbox” package. Click on one of them to display its properties and change the name to ”deselect”.

The last step for this section is to change the logic so that ”deselect” will now look for the unselected state of the checkbox before acting. To achieve this, you actually only need to make one change. Expand the ”deselect” procedure until you locate the ”Check boolean: selected” node. Recall that this is the check-node which queries the state of

the checkbox. The ”select” procedure was set up to throw an exception if the checkbox is not selected. We can reverse that logic now for ”deselect” by simply locating the ”Expected state” checkbox within the node’s properties and unchecking it, as follows:

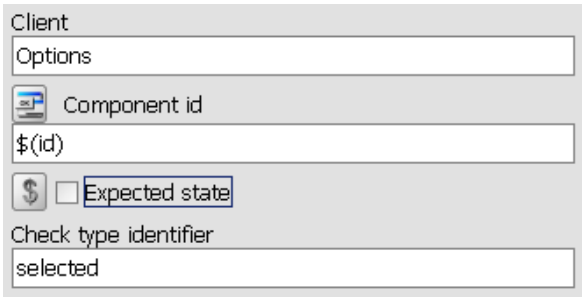

Figure 5.2: Check for Unselected State

Now the check-node is expecting an unselected state of the component and will throw an exception if the component is selected, which is exactly what we want. So that was it! You may want to change some of the labels and or comments within the procedure, for example the name of the ”Try” node could be changed to ”checkbox deselected”. Here’s how your new package should be shaping out:

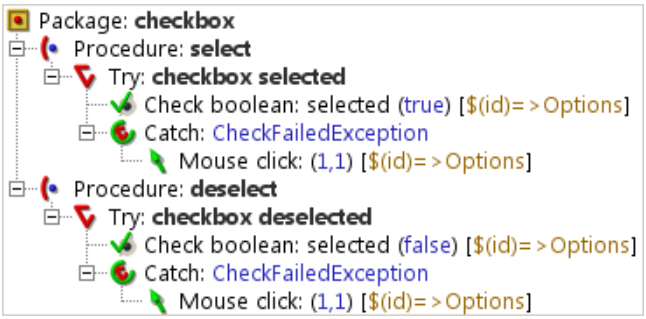

Figure 5.3: Checkbox Package with two Procedures

5.4 Call the Procedures

In the previous chapter (refer to section 4.8[63]) you created a call to the ”selectCheckbox” procedure. Please take a look at the call procedure node now as it has been automatically adapted due to the new package structure and new name.

Calls to procedures within packages follow a syntactical rule which references the procedure’s entire hierarchy in an elegant fashion: `package.procedure` - thus, your procedure call will now be to the procedure named ”checkbox.select”

Note

According to our definition, packages can contain not only procedures but also other packages. If a procedure is located within such a nested package structure, then the same rule for calling the procedure applies as above, with as many `package.` statements before the actual procedure name as needed. For example, your procedure call could be to something like `menu.file.open`, which implies two packages (”menu” and ”file”) with the procedure named ”open”.

Try creating another procedure call now to the ”checkbox.deselect” procedure. You should know how to do this fairly simply by now, using techniques from this tutorial.

5.5 Creating an All-Purpose Procedure

The two procedures you have inside the ”checkbox” package now will allow you to select or deselect a checkbox separately. How would it be if we created one simple procedure which combined both functions? In this and the following sections you will create a procedure called ”setState,” which will set the state of a checkbox to either selected or unselected. This new procedure will include a variable that we’ll call ”select,” which if set to ’true’ will cause the procedure to select the checkbox, otherwise (”select” is ’false’) the checkbox will be deselected.

Let’s start by just creating the new procedure ”setState” within the ”checkbox” package. If you need some assistance on creating a new procedure, you can refer back to section 4.2(55).

5.6 Setting a Default

As we just mentioned, your new procedure will contain the variable ”select” aiming to decide what to do with the checkbox. As with the QF-Test ID which you passed to the procedure in your procedure call, a value of ”select” (either ’true’ or ’false’) must also be passed as an argument to ”setState”.

It is however customary for procedures that make use of such a variable to define a default setting for the variable. This means that passing a value of ”select” to the procedure as an argument will be optional. When no value is passed, the variable will simply take on the value of the default.

Click on the ”setState” procedure node to view its properties in the details window. You’ll notice that included within the properties is an area to define variables. In fact, variables for a procedure are added, edited and deleted in the same way you saw back in section 4.8(63). In that same fashion, add the variable ”select” now and give it the value of ’true.’ Here’s how the properties of your procedure should look:

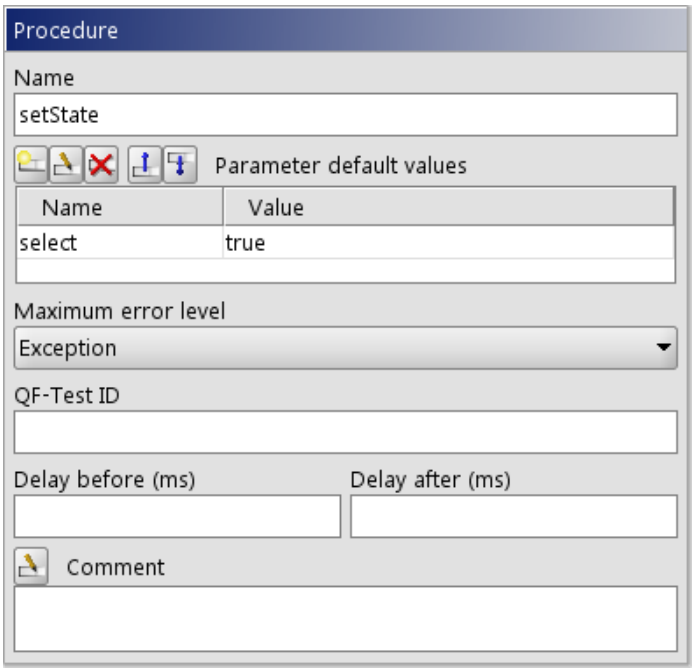

Figure 5.4: Default Value for Procedure Variable

What you've just done is tell QF-Test that if it does not see a value for "select" defined when the procedure is called (i.e., it was not passed to the procedure by the procedure call), then it uses the default value. If a value is supplied as an argument passed to the procedure, then QF-Test will ignore the default and use the supplied value.

Note

For those of you familiar with variables, you'll recognize the concept of the "scope" of a variable. A variable could be, for example, defined in one spot and then used erroneously in another if you forget that it is already defined. On this stage of test-suite development you should not run into this kind of problem, but at some point you may wish to read more about variables in the user manual.

5.7 The If/Else Construct

The variable "select" is intended to help us decide whether to select or deselect a check-box. We now come to the point where we can use this variable inside of the logical if/else construct.

Using the if/else mechanism is quite easy. To start, a condition is given. If this condition evaluates to a true statement, then the nodes under the if-block are executed. Otherwise (the statement is false), the nodes under the else-block are executed.

Our condition is, of course, the value of the "select" variable. If it's set to 'true' then we

select the checkbox, and so on.

Expand and click on the ”setState” procedure node so that you can insert a new node as the first element of the procedure. Now select the Insert→Control structures→If menu option. As usual, you'll be presented with a dialog window to fill in the node's properties. For this node, you need to fill in the condition to evaluate the variable ”select” precisely as shown below:

Figure 5.5: Condition Statement for the ”If” Node

Recall that referring to a variable involves using the `$(variable)` syntax. The single and double quotes in the condition are QF-Test's way of evaluating string literals. Different syntax is required for different types of conditions, such as mathematical expressions. Refer to the user manual's sections on variables and the If control node for further reading.

As we stated, if the condition evaluates to true, then the nodes under the ”If” node are executed. So at this point you can insert the logic that will cause the checkbox to be selected. How do you do this? There are several ways: you could, for example, copy the nodes from the ”select” procedure to this part of ”setState”. However, we'll take the easiest route and just make a call to the ”checkbox.select” procedure here. You'll find help in section 4.8(63) on calling a procedure.

Now we come to the ”Else” portion of the construct. After the last node in your if-block (you'll have just one, but there could be more), you can insert the ”Else” node using the Insert→Control structures→Else menu option. You are not required to fill in any additional information in the properties dialog window that appears for the ”Else” node. It is automatically assumed to be the logical antithesis of the corresponding ”If” node (or nodes - see the **tip** below).

Lastly, we complete the construct by adding the node(s) to be executed when the condition from the ”If” node is not true. So here we simply insert a call to the ”checkbox.deselect” procedure.

The procedure ”setState” is now complete! Here's how it should look:

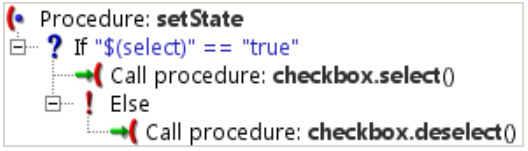

Figure 5.6: The If/Else Construct

Note For our example the boolean nature of the ”select” variable lends itself well to a simple if/else construct. However, for other examples, such as a multi-state condition like the colors of a traffic light, you can implement the more advanced if/elseif/else construct. The concept is the same, you are just permitted now to evaluate multiple conditions. For example, your logic might run like this:

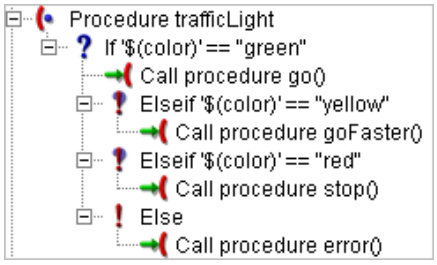

Figure 5.7: The If/Elseif/Else Construct

Note that the ”Else” portion of the if/else or if/elseif/else construct is optional!

5.8 Calling the All-Purpose Procedure

Now you can try calling your new ”setState” procedure. Just create a procedure call node as you did in section 4.8(63). You’ll of course still need the QF-Test ID for the ”id” value as an argument for the procedure call, but now you can also (optionally) add the second variable ”select” to the call, as follows:

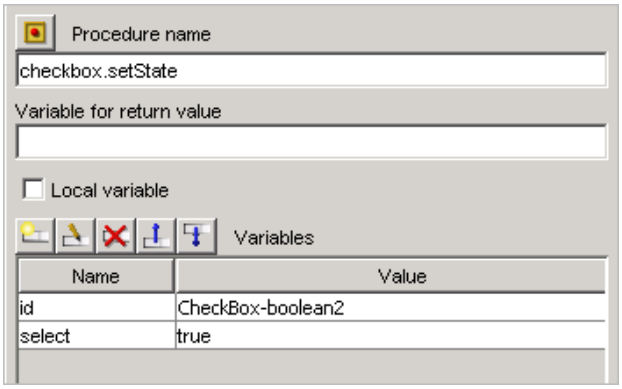

Figure 5.8: Procedure Call to setState

Note Sometimes you may want to create a procedure which returns a value, e.g. a `getState` procedure. For this purpose a Return node can be used inside the procedure. The

returned value may then be assigned to a global or local variable outside the procedure, using the ”Variable for return value” attribute of the ”Procedure call” node, visible in the figure above. For further information please have a look in the reference manual.

5.9 Documenting the Procedure

After you have successfully completed the implementation of your procedure, we want to show a convenient way of how to create documentation for it. For the Java programming language there is a standard way for documenting source code - it’s called `javadoc`. Special tags inserted within comments in the source code are parsed by the `javadoc` tool and provided in a well structured HTML document.

QF-Test offers a similar mechanism to document your packages and procedures. In the following we will show you how it is done. If you want to get a first glance, how a result may look like, please have a look at `qfs_pkgdoc.html`.

Now we want to start to prepare our procedure for the documentation. Please open the ”setState” procedure node if it’s not still open. As you can see in figure 5.9(74) the procedure has two parameters. To get them documented we add two lines in the ”Comment” field of the node, starting with `@param` followed by the name of the parameter and an appropriate description. `@param` is a special tag to indicate the relation of the description to the procedure parameters. Also a general description of the procedure can be added at the beginning of the ”Comment” field. Note that it is possible to use HTML tags for formatting text.

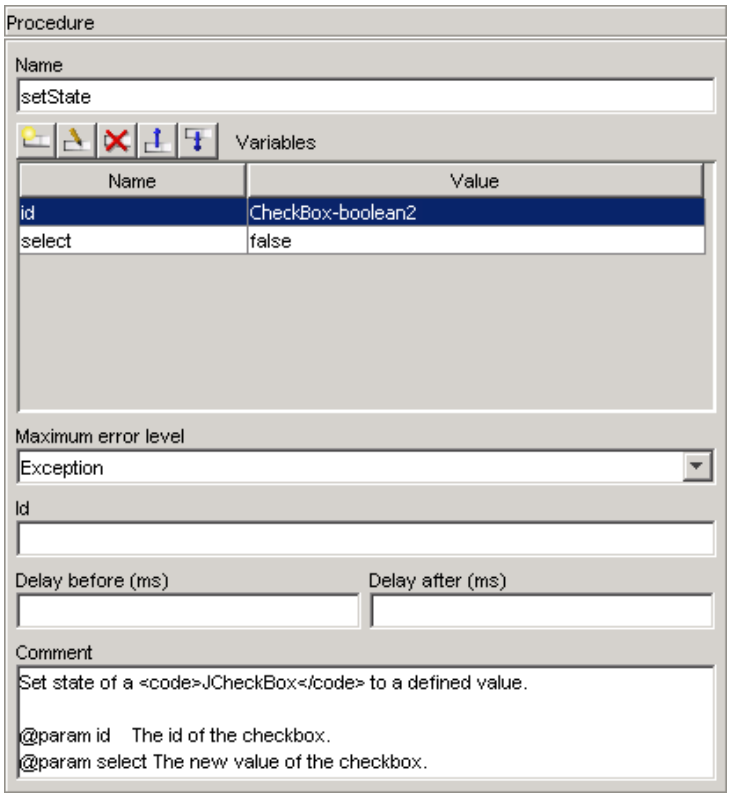

Figure 5.9: Comments for Package Documentation

Please add similar comments to your ”setState” procedure now, so we can start generating the documentation.

The generation process is very easy. Just open File→Create HTML/XML Pkgdoc... from the menu.

You may confirm the upcoming dialog containing some document generation options unchanged by pressing the ”OK” button. Then your default browser should appear automatically (you may have to switch to it in case it’s already open an doesn’t pop to the foreground) showing the following.

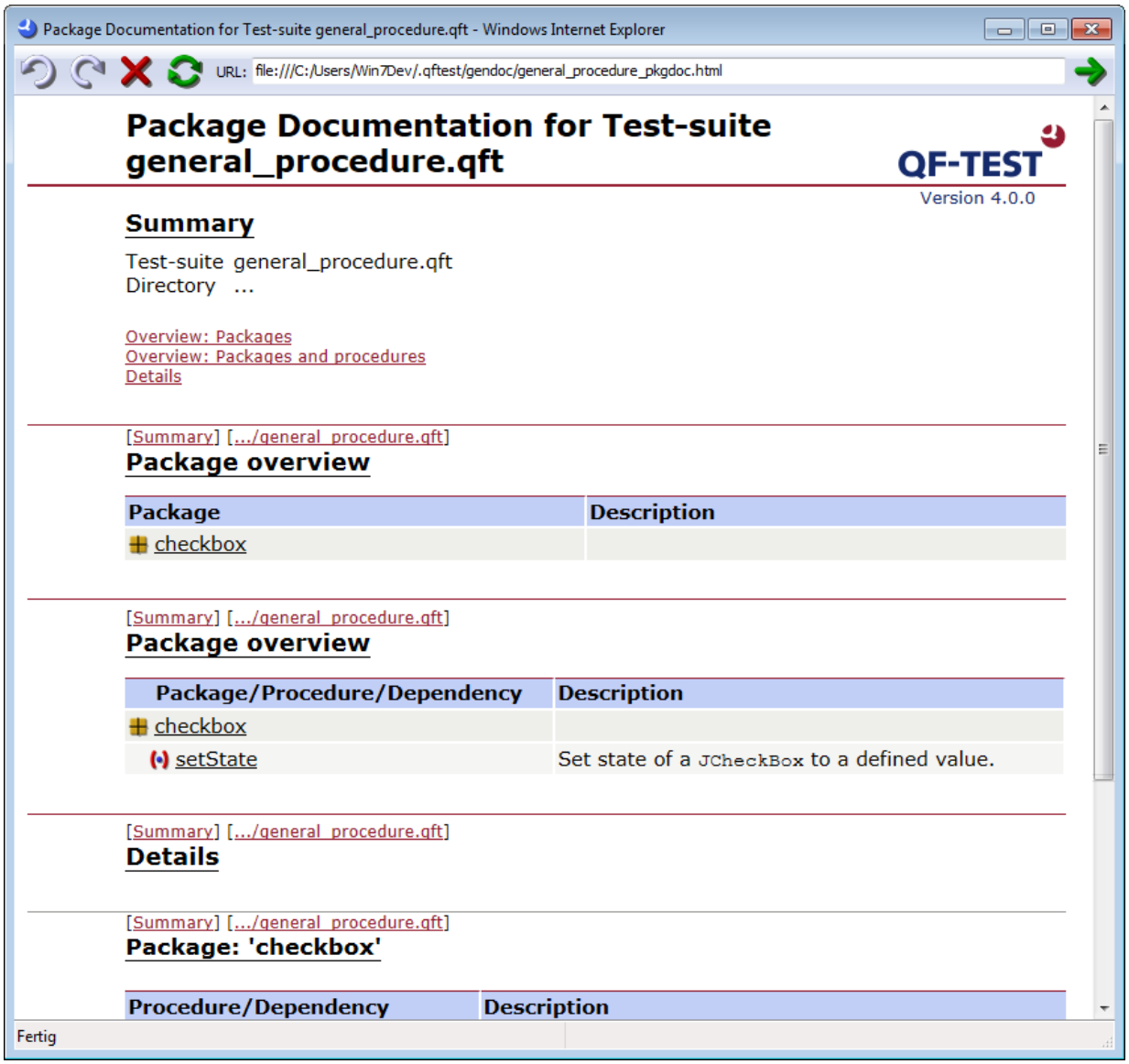

Figure 5.10: The Package Documentation

Here we see three parts in the document. The package overview, procedure overview and the detailed description of the package ”checkbox”. In the package ”checkbox there is the procedure ”setState” with its general description text and its parameters.

With the package documentation feature you can easily generate a helpful description of packages and procedures, especially if you want to provide general functionality within a kind of library. Such a library including documentation also comes with QF-Test. It is explained in detail in chapter 7[83].

Now you can play a little around with the documentation generation. You may e.g. add

a general description to the package ”checkbox” and try additional tags like @version, @author, @result, @throws. For a full list of tags please have a look at the user manual.

Note You can also use QF-Test to create the package documentation automatically in batch mode. Please see the user manual for details about the relevant command line arguments.

5.10 Saving your Test-suite

At this point, you are well on your way to advanced development of test-suites. You may want to take the time now to save the work you've done thus far on the test-suite. Saving a test-suite can be done simply with the File→Save menu option. You can name the test-suite as you like, but we suggest the name of `utils.qft`, as we will be referring back to and modifying this test-suite in the next chapter.

NOTE If you are evaluating QF-Test in the demo-mode, you'll not be able to save the test-suite. For a full-featured evaluation license, we refer you to the Quality First Software GmbH download page[1].

In the next chapter, we'll begin to explore even more capabilities of QF-Test by making a modularized library of your `utils.qft` test-suite. This is, however, also a good breakpoint if you wish to explore features of QF-Test on your own (with the help of the user manual!) until you are ready to come back to the tutorial.

[1] http://www.qfs.de/en/qftest/download.html

Chapter 6

Modularization [30-45 min]

In this chapter we will expand upon the concepts you learned in the previous chapters. Our goal is to show you how you can modularize test-suites. The features we'll be presenting to you here are a bit more advanced than those seen in earlier chapters, so we strongly recommend that you have gained a solid understanding of the concepts already covered before proceeding.

As you begin to develop test-suites specific to your own needs, you'll often run into situations where you are repeatedly, or perhaps redundantly, performing the same steps. Suite A, for example, may often require the usage of some menu operation in your SUT that Suite B also needs. Instead of having the same procedure in both suites, it is much more useful - and reliable - to have one common source for the procedure.

This brings us to the concept of a utility suite - or library - which contains common procedures for use by any number of other test-suites. Your `utils.qft`, which you hopefully saved in the previous section 5.10(76), will then serve as a sort of library that other test-suites can access for common utility procedures. The advantage of such a structure is that you have a stable library of routines which can be relied upon to perform their function accurately by other test-suites. Such modularization of suites is easy to implement in QF-Test, as will unfold in the following sections.

6.1 Creating the Driver Suite

Open a new suite now with the File→New Suite menu option. With this test-suite you'll create a real test for the SUT, as we'll show in the following sections.

You should also bring up your `utils.qft` test-suite, which contains the "checkbox" package you wrote in the previous chapter.

Save the new driver suite in the same directory as `utils.qft`. This is necessary so

you can use relative path names when one test-suite refers to the other. The actual filename doesn't matter.

6.2 What Belongs Where

If the SUT application is not already running, start it up now. You'll note that the startup nodes for the SUT are still located in `utils.qft`. A true utility suite should have no direct connection to an SUT. Why? Because the utility suite could be used by other test-suites that test different SUTs.

For example, say you have two completely different SUTs 'A' and 'B' which you must test. You would thus create two test-suites for each SUT. However, in both SUT applications, checkboxes are present that can be selected or deselected using your `utils.qft` utility suite. `Utils.qft` must therefore be general-purpose enough to handle this type of situation.

As it is now, `utils.qft` contains elements that directly link it to the Options Demo SUT, which we created back in section 4.1[52], i.e., the Setup/Cleanup nodes as well as the "Windows and components" section. This SUT-specific information is now irrelevant to the utility suite and should be moved over to the test-suite which will be directly testing the SUT.

So, similar to the steps done in section 4.1[52], you can now move the Setup/Cleanup nodes and the contents of the "Windows and components" node to your new suite. Make sure to use the Cut menu option instead of Copy so that the nodes are completely removed from `utils.qft`. When you're done, your new test-suite should look just like the skeleton we saw in figure 4.1[53]. You can also clean out the procedure call nodes you created in `utils.qft` so that it is down to the bare essentials as shown in the following figure:

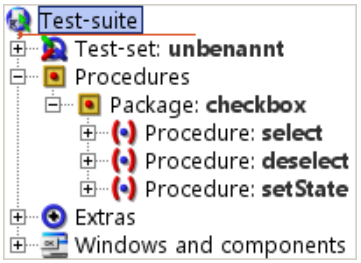

Figure 6.1: Utility Suite `utils.qft`

6.3 Creating a Test Sequence

In this section, you'll use your new test-suite to create a test sequence for the SUT. The test sequence will be a simple one: select the "Miscellaneous" window of the SUT and deselect the "Boolean option" checkbox.

After the Setup-node in your test-suite, insert a sequence-node with the Insert→Sequence nodes→Sequence menu option. Give it a name such as "Deselect Checkbox". Expand the node so that you can insert new nodes into it.

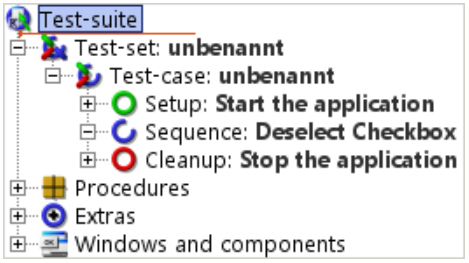

Figure 6.2: Deselect Checkbox Sequence-Node

Now click on the record button of your test-suite so that we can record a sequence. Bring up the window of the SUT and click on the "Miscellaneous" item under the "Preferences" list. You should see the "Miscellaneous Options" sub-window appear in the SUT. Return to your test-suite now and click the button to stop recording.

In the recorded sequence that appears in the your "Extras" node, you should just see just one event recorded, namely the click of the "Miscellaneous" item. Move this node into your new sequence-node, which should now look something like this:

Figure 6.3: Sequence-Node with Recorded Click

6.4 Calling a Procedure in the Utility Suite

The next step in the test sequence is to make a call to the utility suite to deselect the checkbox. As you did in section 5.8[72], insert a procedure call node to the procedure "checkbox.setState". This time, however, we need to add a slight modification, since the procedure is located in another test-suite.

Procedure calls in QF-Test can be prepended with the name of the suite in the form of `suite#procedure` to indicate that the target procedure is located in the supplied test-suite. For our example, the procedure call is correctly declared `utils.qft#checkbox.setState`. Don't forget to fill in the variables as we did in section 5.8[72], with "select" set to 'false.'

Figure 6.4: Procedure Call to utils.qft

6.5 Adding an Include

Instead of making procedure calls containing an explicit reference to the test-suite such as with `utils.qft#checkbox.setState`, QF-Test provides a simple mechanism which allows you to 'include' certain test-suites as part of your overall test-suite structure.

An 'include' is an implicit reference to a test-suite. When QF-Test sees a procedure call node without an explicit reference to a test-suite (i.e., there is no test-suite name prepended with the '#' symbol before the name of the procedure), then it attempts to search within the current suite for the procedure being called. If the procedure is not found in the current suite, it then searches the test-suite's 'include list' for the procedure.

Click on the "Test-suite" root node of your test-suite. Within the properties of this node, you'll see a section for "include files". Within this list, add now the name of your utility suite, `utils.qft`. Entries are added and modified in this list just like you did with variables, using the small add, edit and delete buttons. You should see something like this:

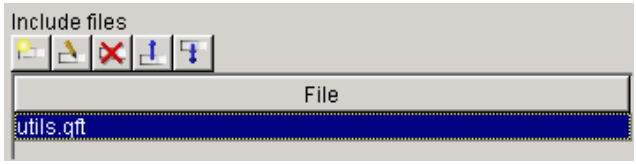

Figure 6.5: An Include List

Note For more detailed information about how QF-Test resolves issues concerning procedure calls, refer to the include file resolution section of the user manual.

6.6 Modularization for Multiple SUTs

If you recall the discussion back in section 6.2[78], a utility suite such as `utils.qft` should not contain any direct references to the SUT client. If you scan through `utils.qft`, however, you'll see that there are indeed still such references to the Options demo. Take a look at one of your "Check boolean: selected" nodes, for example, and you'll see the reference to the client as such:

Figure 6.6: Constant Reference to the SUT Client

This direct reference will, of course, prevent you from using `utils.qft` with other test-suites, but the solution is simple. Instead of directly referencing the SUT, you will replace the reference with a variable that can (and must) be set by the test-suite which utilizes `utils.qft`.

So let's fix this now. Where-ever you see a reference to the "Options" client, replace it with a variable reference, let's call it `$(client)` as shown below:

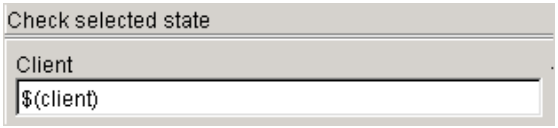

Figure 6.7: Variable Reference to the SUT Client

You'll have to change all references to the client in `utils.qft`, but QF-Test can make this a lot easier for you by using the search/replace feature. Click on the top-level "Test" node of the test-suite and select the Edit→Replace menu function.

The next step is, as you may suspect, to modify the driver test-suite so that a value of the variable "client" is available when procedures inside of `utils.qft` are used. One way to do this is to pass the "client" variable as an argument for each procedure call into `utils.qft`. This kind of solution is perfectly viable, if not a little manual-intensive and error-prone.

Another more elegant solution is to set the variable once in your test-suite so that you don't have to think about it again. Expand your top-level "Test-suite" node. Within the properties window of the test-suite, you will see a section (beneath the includes and

dependencies) in which you can define variables for the test-suite. Within this area, add the variable `client` and give it the value of `Options` like this:

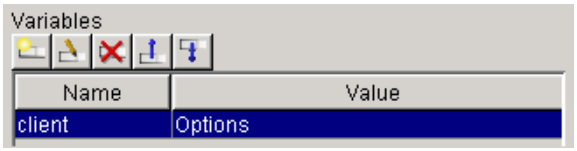

Figure 6.8: Setting a Default Test-suite Variable

Once you've completed this step, you shouldn't have to worry about this variable again until you create a new test-suite that uses `utils.qft`. Your utility suite is now fully modular!

Chapter 7

The Standard Library [30-45 min]

We complete the concept of modularization in this chapter by presenting a standard library of utility functions. Some of the utilities you'll already be familiar with, like operations for dealing with checkboxes. Other utilities will be new. The following sections will cover in detail each of these operations.

There are several things to be learned from this chapter. Most importantly, you'll see a workable utility library (in fact, the library evolved from a very similar library used for regression tests of QF-Test at Quality First Software GmbH). In the process of analyzing the operations contained within, you will hopefully obtain ideas of how to better modularize your own test-suites and simplify the overall testing process.

The procedures within this library should work with any standard Java/Swing or Eclipse/SWT application; that is, they were designed independently of any specific SUT. Thus, they should fit the needs of your SUT quite nicely. The library is contained in the file `qfs.qft` and is included as part of the QF-Test distribution.

To make use of `qfs.qft` click on the "Test-suite" root node of your test-suite. Within the properties of this node you'll see a table for "Include files". Please add `qfs.qft` to this list. A path information is not necessary as the `include` directory of QF-Test is contained in the library path (see also Reference part of the manual). Now you can select procedures of this library at any procedure-call.

In addition to the description provided in this tutorial you can find HTML documentation of the standard library in a `javadoc`-like format. The file named `qfs_pkgdoc.html` is located in the directory `qftest-4.0.8/include/qfs_pkgdoc`.

7.1 The SUT For Testing

As we've already noted, the standard library to be introduced is not SUT-specific. However, for purposes of demonstration within this tutorial we will use an application specif-

ically tailored for use with the standard library.

Bring up the test-suite `StdLibDemo.qft`, which - like the suite `Options.qft` you should be familiar from earlier chapters - is located in directory `qftest-4.0.8/doc/tutorial` from your QF-Test installation. You should see the following:

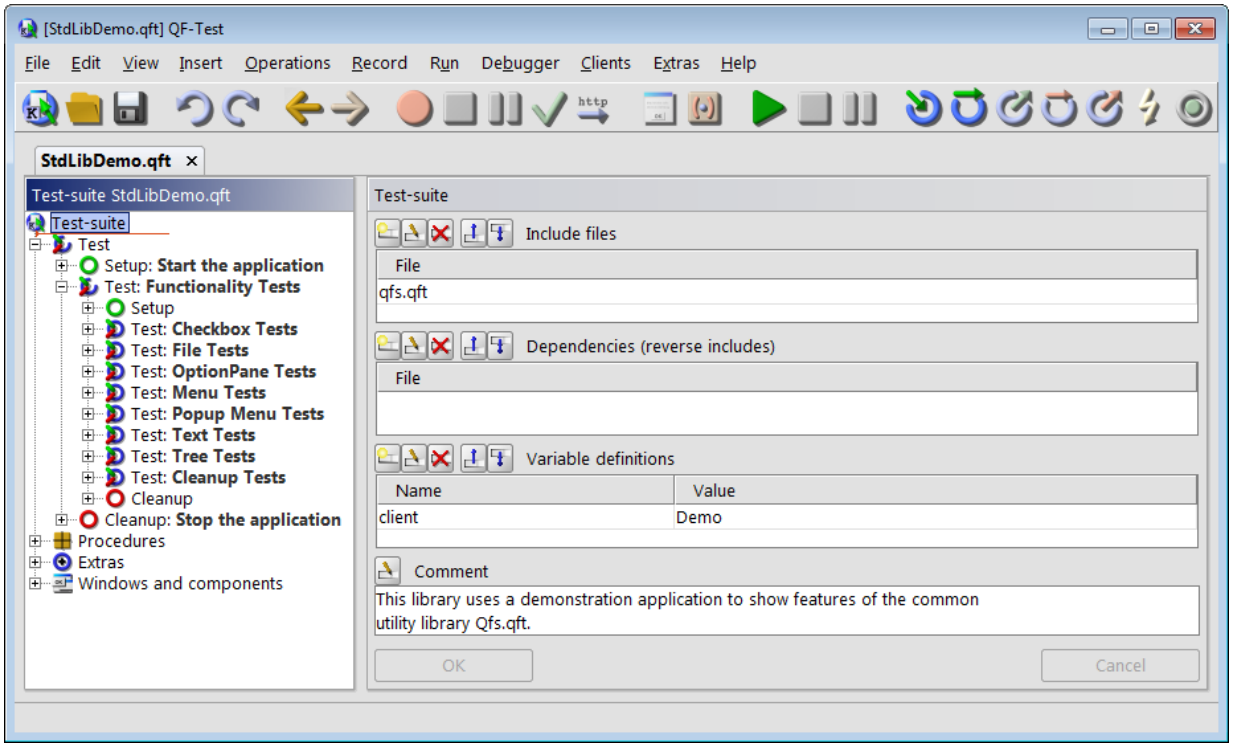

Figure 7.1: The Test-suite `StdLibDemo.qft`

Start the SUT application by executing the Setup node (as seen). In a few moments, the window of the SUT should appear. If not, it is probably hidden beneath the test-suite.

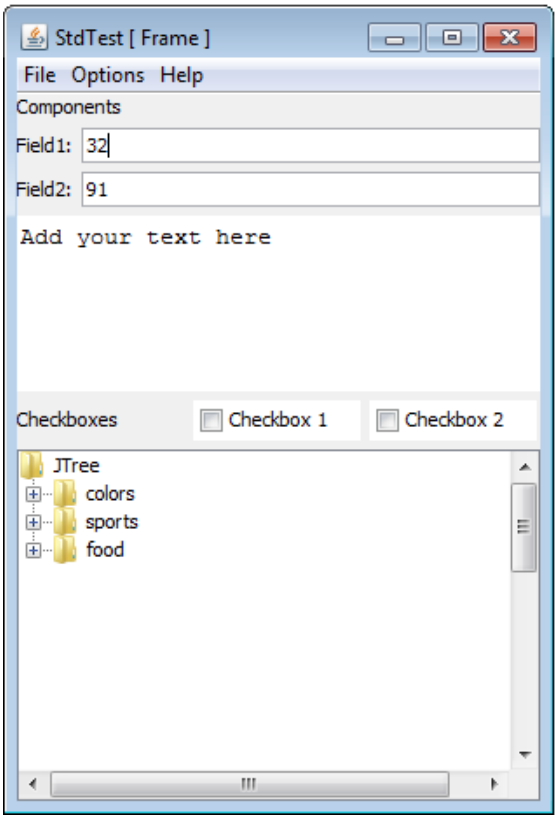

Figure 7.2: The SUT for Testing the Standard Library

7.2 The Standard Library

Locate and load the test-suite file `qfs.qft`, which is located in the `qftest-4.0.8/include` directory of your QF-Test installation.

Here we see a brief overview of the packages available:

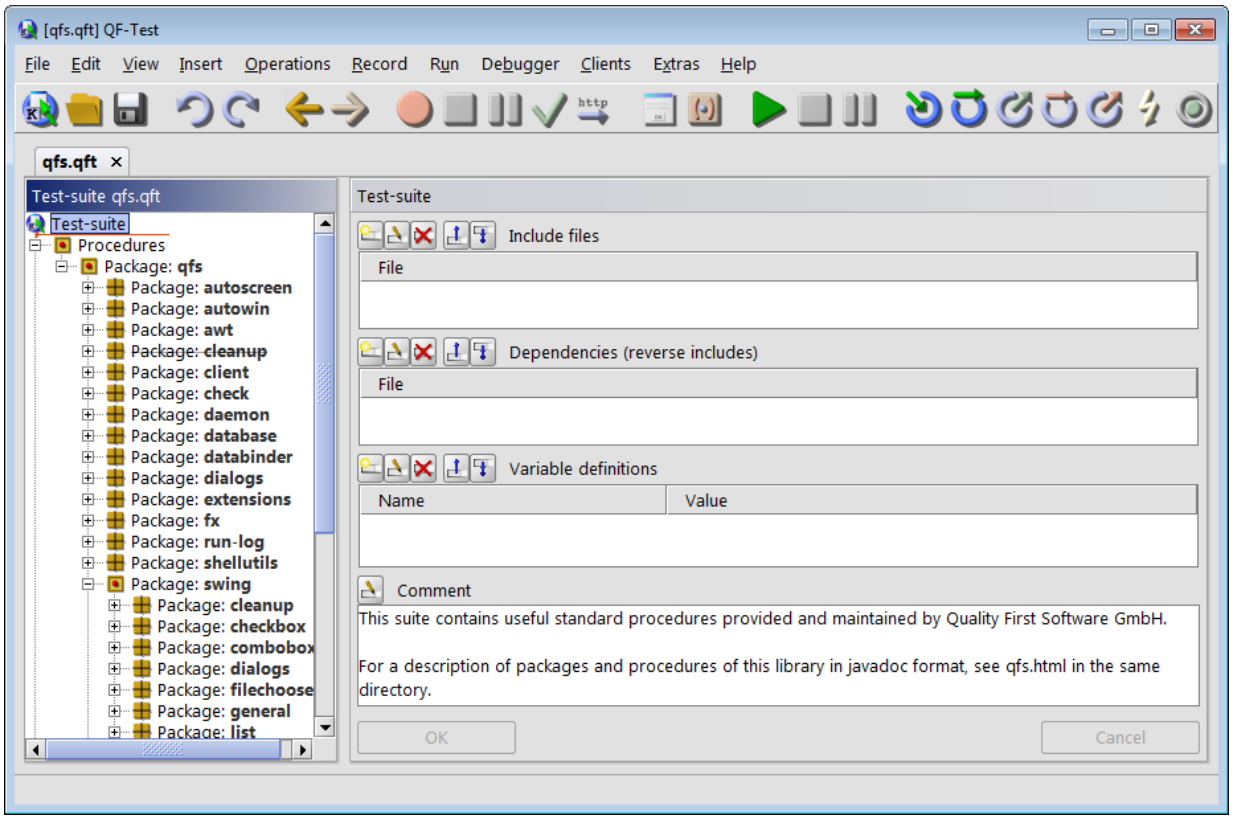

Figure 7.3: The Standard Library

The suite primarily contains procedures for dealing with Java/Swing and Eclipse/SWT components. The ”swing” and the ”swt” package contain nearly the same procedures to deal with several components. The structure of both packages is the same, so the following description will cover the Java/Swing as well as the Eclipse/SWT procedures. There are also a couple of other usefule packages included, which are also documented below.

Within all of the procedures of this library, you’ll notice that the variable `$(client)` is referenced. This is a standard mechanism that you’ll often see repeated in many test-suites for creating independence from a specific SUT. Here, the library assumes that the test-suite which uses the library will set a value for `$(client)` prior to using any procedures. For example, here we see the properties of the ”Test-suite” root node of the suite `StdLibDemo.qft`, in which we set down the value of `$(client)` as a suite variable. Thus, any nodes executed within the scope of this test-suite (such as the procedure calls to `qfs.qft`) will have access to the variable.

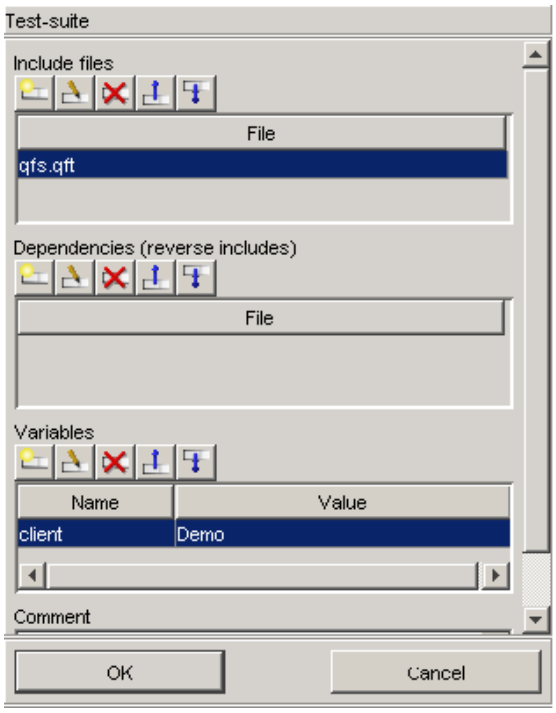

Figure 7.4: Setting the $(client) Variable

You could alternatively set the value of `$(client)` per call to a procedure within `qfs.qft`. Such a scheme might be useful when you have more than one SUT. Just to hammer this point down, please note that `qfs.qft` cannot have any default value set for the `$(client)` variable as this would undesirably couple it to a specific SUT. It is up to you to set a value of the `$(client)` variable for the standard library to use!

One further note before we continue: you may have noticed from the figure above that we've included `qfs.qft` within the demonstration test-suite without any direct reference to the directory in which the file is located. This is because the `include` directory of your QF-Test distribution has been added to QF-Test as an implicit library path for all of your test-suites. This means that any test-suite file located within this directory can be included within your own test-suite. For further library paths, look at the global options for QF-Test under Edit→Options→General→Library.

A complete description of all packages and procedures including parameters and return values is given in the library's HTML documentation, also accessible from the QF-Test Help menu. The latest version is even available online at http://www.qfs.de/include/qfs_pkgdoc/qfs_pkgdoc.html.

7.3 Selected Packages and Procedures

We will now have a closer look at a number of selected packages and procedures from the standard library.

Please note, that following examples are mainly based on the `qfs.swing` procedures. The procedure of `qfs.swt` can be used in the same way. Procedures, which are Java/Swing or Eclipse/SWT specific will also be handled at the end of this chapter.

7.3.1 The Checkbox Package

We begin by looking at the `swing.checkbox` and `swt.checkbox` package, which are a good starting point as the procedures here should look very familiar if you developed the `utils.qft` library suite in chapter 6(77).

The procedures available within this package are:

- **select** Selects (checks) a checkbox. If the checkbox is already selected, then no action is taken.
- **deselect** Deselects (un-checks) a checkbox. If the checkbox is already deselected, then no action is taken.
- **set** Sets a checkbox to a given state (true or false).

For each of these procedures, you pass the QF-Test ID of a checkbox component as a variable argument. The library handles verification of whether or not the checkbox state was properly set as expected. To see example usage of these procedures, look in the ”Checkbox Tests” node of `StdLibDemo.qft`.

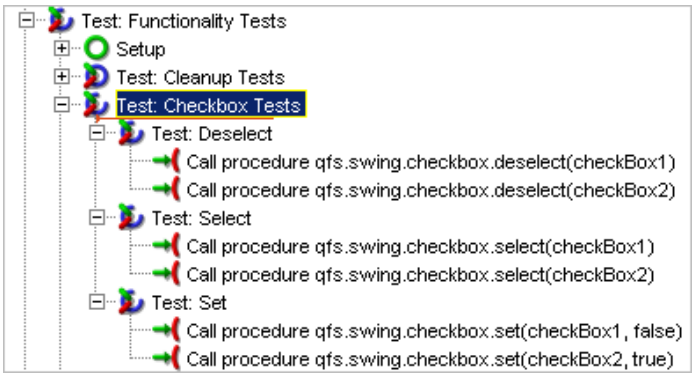

Figure 7.5: Example Usage of the `qfs.swing.checkbox` Package

Usage of the other procedures in this package follows the general model seen here.

7.3.2 The Combobox/Combo Packages

The packages `swing.combobox` bzw. `swt.combo` contain procedures to select a value in a combobox.

The procedures available within this package are:

- **setValue** Select a value in the list of the combobox.
- **setValueViaSUTScript** Select a value in the combobox using a SUT-script. It uses the method `setSelectedItem()` for Swing and `select()` for SWT.

7.3.3 The General Packages

The packages `swing.general` bzw. `swt.general` contain useful procedures to work with components.

The procedures available within this package are:

- **clickAtComponent** Perform a mouse-click on a given component, but only if an according parameter is set.
- **doClick** Perform a mouse-click on a given component.
- **setLocation** Set the location of a given component.
- **setSize** Set the size of a given component.

7.3.4 The List Packages

The packages `swing.list` bzw. `swt.list` contain useful procedures to work with lists.

The procedures available within this package are:

- **getItemCount** Return the number of items of the list.

7.3.5 The Menu Package

The `swing.menu` and `swt.menu` packages allow you to easily select items and checkbox items from menus or sub-menus. The procedures visible after expanding the package node are:

- **selectItem** Selects an item from a menu.
- **selectSubItem** Selects an item from a sub-menu.
- **setCheckItem** Sets the state of a checkbox menu item (true or false).
- **setSubCheckItem** Sets the state of a checkbox item from a sub-menu (true or false).

For each of these procedures, you must pass the QF-Test ID of the menus as well as the item and/or sub-item to select or check; the usage varies slightly depending on the nature of the procedure. Take a look at the example usage "Menu Tests" node. For example, a call to "setSubCheckItem" looks as such:

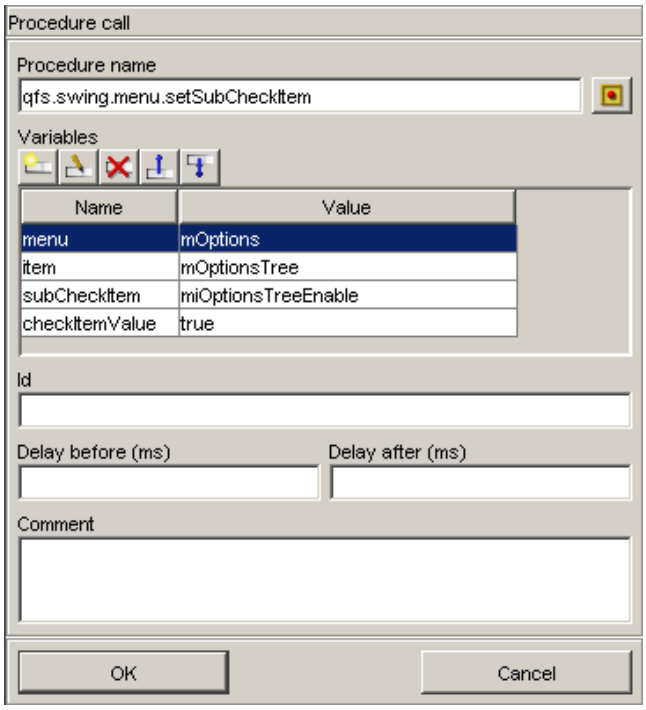

Figure 7.6: Example Call to `qfs.swing.menu.setSubCheckItem`

Here we see the variable arguments needed to implement the procedure call. These arguments are organized as:

```
menu -> item -> subCheckItem (checkItemValue)
```

which represent the following implementation within the SUT:

```
Options -> Tree -> Enable (true)
```

Clearer may be the actual process as seen within the SUT:

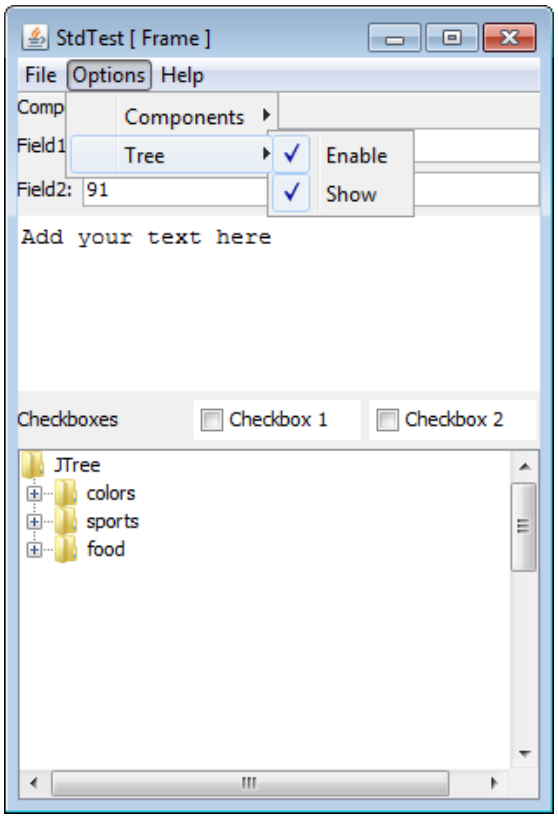

Figure 7.7: Selecting a Sub-Menu Check Item within the SUT

7.3.6 The Popup Menu Package

Popup menus are similar to regular menus in that they contain items, sub-menus and checkbox menu items. The primary difference is the means by which a popup menu is invoked, which is by right-clicking a component that has been configured to display a popup menu.

The packages `swing.popupmenu` and the `swt.popupmenu` contain the same procedures as `swing.menu` and `swt.menu`; only the implementation is different. For each procedure, your test-suite must first take care of opening the popup menu before calling any procedure from the `swing.popupmenu` or the `swt.popupmenu` package, as a popup menu is component-specific. Consider the following example as shown within the SUT:

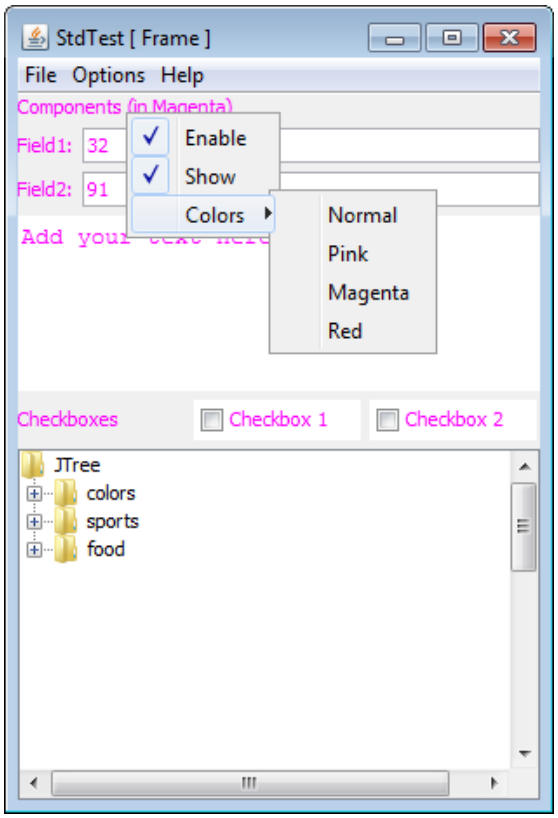

Figure 7.8: Selecting a Sub-Menu Item within a Popup Menu

We implemented this example through the following test in `StdLibDemo.qft`. The first node brings up the popup menu, and then the call to the procedure is made, followed by a check of the results.

Figure 7.9: Example Call to `popupmenu.setSubItem`

7.3.7 The SWT/Sash Package

The package `swt.sash` contains procedures for working with components of the class `org.eclipse.swt.widgets.Sash`.

- **moveSash** Move a sash by a given distance.

- **moveSashTo** Move a sash to a given position.

7.3.8 The Table Package

The packages `swing.table` and `swt.table` provide utility procedures for tables.

- **getRowCount** Return the number of rows of a table. For Swing it uses the method getRowCount() and for Eclipse/SWT it uses the method getItemCount().
- **getColumnCount** Return the number of columns of a table. It uses the method getColumnCount() for Swing and Eclipse/SWT
- **resizeColumn** Resize the width of a given table column. **Note:** At the moment the Eclipse/SWT implementation isn't available.
- **selectCell** Select a given table-cell.

The trick in the implementation is the "Fetch Geometry" node to get the current column width. From this position the resize action is performed by dragging the mouse appropriately to the left or to the right.

7.3.9 The Table/Selection Package

The packages `swing.table.selection` and `swt.table.selection` provide procedures to do selections on certain table cells.

- **deselectAllRows** Deselect all rows.
- **deselectRow** Deselect a certain row specified by index.
- **deselectRowRange** Deselect a range row of rows specified by index.
- **selectAllRows** Select all rows.
- **selectRow** Select a certain row specified by index.
- **selectRowRange** Select a range row of rows specified by index.

The trick in the implementation is the "Fetch Geometry" node to get the current column width. From this position the resize action is performed by dragging the mouse appropriately to the left or to the right.

7.3.10 The Tabbedpane/CTabfolder Package

The packages `swing.tabbedpane` and `swt.ctabfolder` provide utility procedures for tabbed-panels or tab-folders.

- **closeTab** Close a tab via clicking at at the ”X” button. This might not work with all implementations of TabbedPanes and CTabfolder.
- **selectTab** Select a tab in a given ctabfolder.

7.3.11 The Text Package

The packages `swing.text` and `swt.text` provide utility procedures for text fields and text areas, the need for which you may have already come across while developing test-suites.

- **setText** Set the value of a specified textfield.
- **clearField** Clear a text field. This procedure cannot be used with a text area.
- **clearArea** Clear a text area. This procedure cannot be used with a text field.

Note

These procedures are now almost redundant, because ”Text input” nodes can clear the target component as needed. However, the procedures might still come in handy now and then.

For each of these procedures, simply pass the QF-Test ID of the text field/area component. Here we see a test within `StdLibDemo.qft` that uses makes use of the text area component within the SUT. In this test, we first clear the text area and input two lines of text. Then the text area is verified to confirm that it contains the expected test. Following this, we clear the text area again and verify that it is empty.

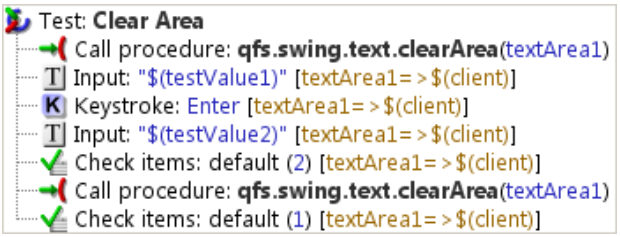

Figure 7.10: Example Usage of `text.clearArea`

7.3.12 The Tree Package

We've provided some simple access procedures for manipulating trees within the packages `swing.tree` and `swt.tree`. These include:

- **collapse** Collapse a node of a tree.
- **collapseNode** Collapse a node of a tree using separate parameters.
- **expand** Expand a node of a tree.
- **expandNode** Expand a node of a tree using separate parameters.
- **selectNode** Select a given tree-node.

For any of these procedures, you simply pass the QF-Test ID of the node you wish to manipulate. For example, here we see a call to the procedure to expand a tree node:

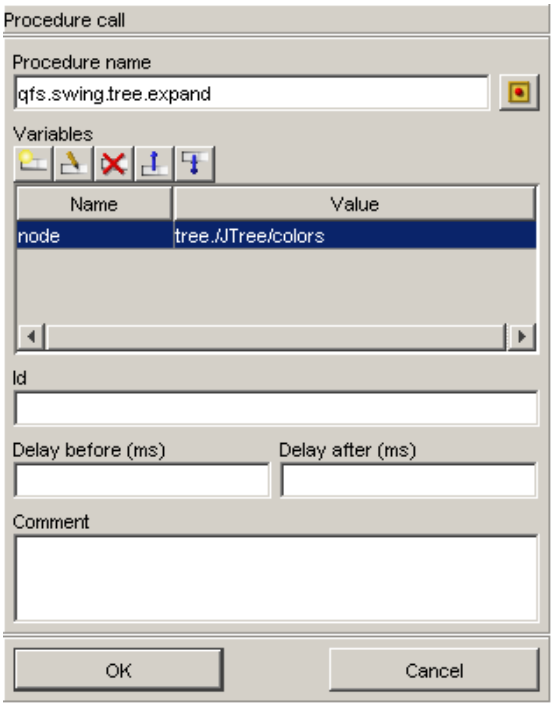

Figure 7.11: Example Usage of `tree.expand`

Note that in this example, we use a component of a tree node which we recorded earlier. If you look in the "Windows and components" node of the `StdLibDemo.qft` test-suite, you'll see the individual tree-item components we recorded:

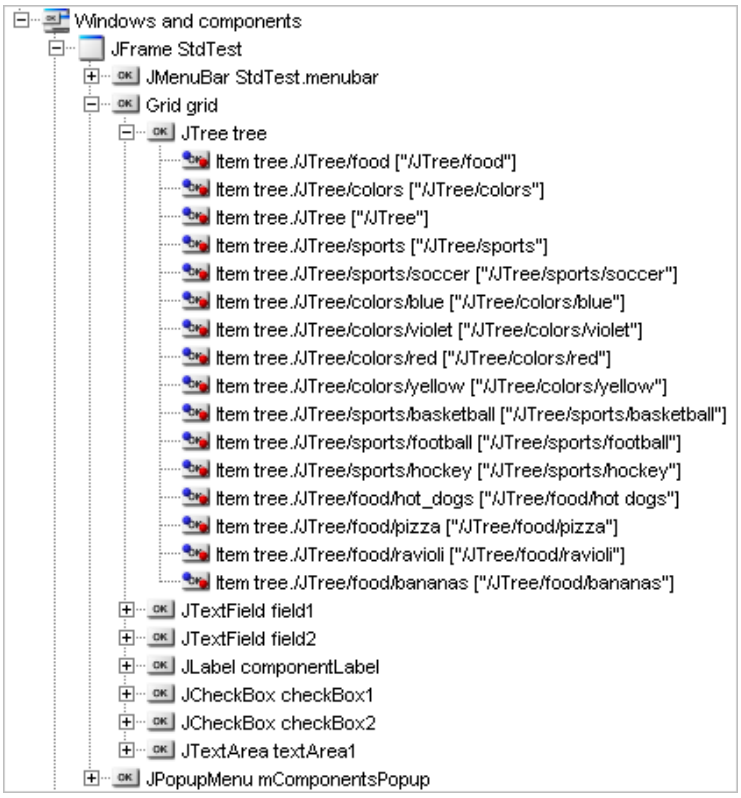

Figure 7.12: Tree Component Items

7.3.13 The Cleanup Package

The packages `swing.cleanup` and `swt.cleanup` are useful for generic cleanup of the SUT environment after an unexpected exception occurs. Imagine, for example, that an exception is thrown while attempting to manipulate a menu in the SUT. The exception will cause the execution path within your test-suite to be re-directed to an exception handler, or an ”implicit” exception handler. This means that the normal flow of execution, which would have properly closed the open menu, has now been interrupted. Without proper action, that menu could be left open and thus block other events directed to the SUT.

Here is the list of available procedures within this package:

- **closeAllModalDialogs** Ensure that modal dialog windows of the SUT are closed. **only available for Swing**!
- **closeAllDialogs** Ensure that all dialog windows of the SUT are closed. **only available for Swing**!

- **closeAllDialogsAndModalShells** Ensure that dialog and modal shells of the SUT are closed. **only available for Eclipse/SWT**!
- **closeAllMenus** Close all menus of the SUT unconditionally.
- **implicitExceptionHandler** Use as a generic handler for dealing with implicitly caught exceptions.

The first procedures should be relatively clear in meaning and purpose. The procedure `implicitExceptionHandler` in fact encompasses the usage of the other procedures within this package, which makes it a good default handler for your own test-suite should unexpected exceptions arise.

The concept of implicit exception handling is an important one as an exception in a single test-case is not meant to stop the whole test-run. Just the current test-case needs to be aborted and then continue with the next test-case.

Therefore, an exception raised during the course of a Test-case will be caught at that level and not propagated to prevent aborting the whole test-run. The error state is duly noted in run-log and report however.

If the Test-case uses a Dependency the exception is passed to the Catch node of the same, if available. This kind of exception (and error) handling is described in the Dependency chapter of the manual.

7.3.14 The Swing/Filechooser Utility Package

The package `qfs.swing.filechooser` presents a slight variation on the utility procedures we've seen in this library up to now. This package is intended not only to assist in manipulating any file chooser dialog (the JFileChooser swing component, to be specific), but also to help in recording and identifying components of this dialog window. We'll first list the procedures and then get into more detail:

- **selectFile** Select a file within a file chooser dialog window.
- **enableNameResolver** Installs the JFileChooser name resolver (more below).
- **disableNameResolver** De-installs the JFileChooser name resolver.

The the first procedure (selectFile) is fairly straight-forward. You simply provide the file name that will be input to the "file name" text field (or whatever it may happen to be called) of the dialog. Your test-suite must first handle opening the dialog. For example:

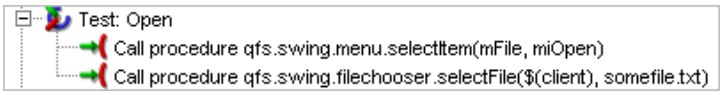

Figure 7.13: Example Usage of `qfs.swing.filechooser.selectFile`

However, using `selectFile` will **not** work until your test-suite enables the name resolver! We come to that next.

The procedure `enableNameResolver` installs and enables an extension to QF-Test called the JFileChooserResolver. This name resolver will help you tremendously in dealing with the short-comings of the Java JFileChooser dialog. It is a standard `NameResolver` as explained the chapter named "Name resolver hooks" in the technical reference manual.

The interesting thing with JFileChooser is that components within the standard dialogs (and indeed the dialog window itself) have not been named with the Java `setName` operation. This can lead to some confusion in developing test-suites, as the same components appear to show up in different dialog windows. For example, the "File Name" text field will be seen in both a "Save" and an "Open" file chooser dialog.

For QF-Test this is not a problem. It has powerful algorithms for recording and recognizing components and therefore should not confuse an unnamed and non-unique component that appears in multiple windows. The confusion will arise for the test-suite developer mostly in terms of managing the components. Take a look at the component list recorded for the two "Open" and "Save" file chooser dialogs for the demo application:

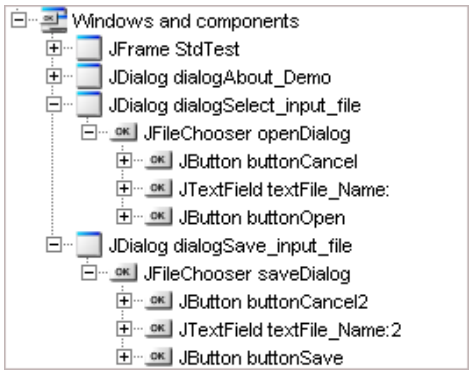

Figure 7.14: Recorded JFileChooser Dialogs and Components

As you can see, QF-Test correctly identified the components and gave each a unique identifier, such as "textFile_Name:2" - which represents the second of the same component of this type that was seen. Imagine now that you have many other kinds of JFileChooser dialogs, each with its own list of components that must be separately

maintained. Writing test-suites under this type of environment can quickly become unwieldy.

Enter the name resolver. This is a feature which enables QF-Test to assign names to unnamed components of a JFileChooser dialog as if they had been named within the original Java source code using `setName`! The result is that the JFileChooser and its components can be generically treated. That is, you'll need just one list of components for any number of JFileChooser dialogs that may be manipulated within your test-suite. Here we see an example usage:

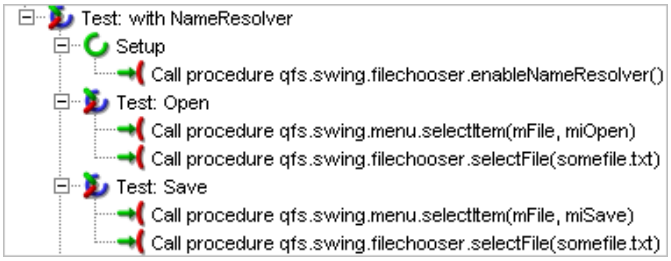

Figure 7.15: Using the Name Resolver

We see in the implementation of `selectFile` that the generic components are referenced:

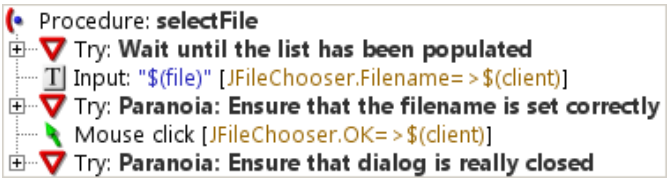
Procedure: selectFile
Try: Wait until the list has been populated
Input: "$(file)" [JFileChooser.Filename=>$(client)]
Try: Paranoia: Ensure that the filename is set correctly
Mouse click [JFileChooser.OK=>$(client)]
Try: Paranoia: Ensure that dialog is really closed

Figure 7.16: Implementation of `selectFile` with Generic Components

Indeed for your own test-suite, you should not even need to maintain any components for any JFileChooser dialogs, if you are going to use the name resolver. The generic components are stored within `qfs.qft`:

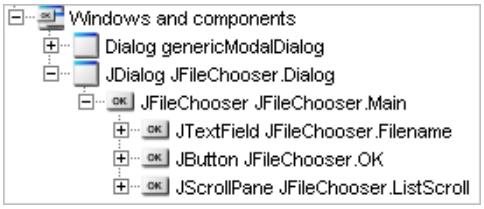

Figure 7.17: Generic FileChooser Components in `qfs.qft`

Note that within our demonstration test-suite `StdLibDemo.qft`, the name resolver is enabled and disabled during the setup and cleanup. This is really just overkill and only done for demonstration purposes. You should only need to enable the name resolver once in your test-suite and leave it be. There's also no direct need to disable the name resolver after it has been enabled.

7.3.15 The Swing/Optionpane Utility Package

The package `qfs.swing.optionpane` contains utility procedures to access dialogs of type `JOptionPane`. `JOptionPane` is the swing component used to create those little message boxes informing the user about program errors, warnings or the need for confirmation.

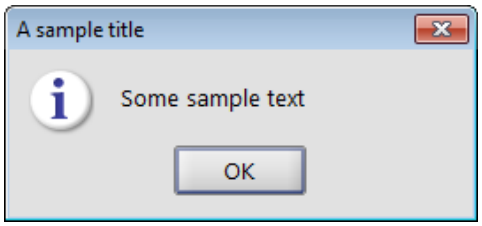

Figure 7.18: `JOptionPane` dialog

The first group of procedures allows identifying the type of the dialog:

- **isErrorDialog** Checks whether the dialog is a Error dialog.
- **isInfoDialog** Checks whether the dialog is a Info dialog.
- **isWarningDialog** Checks whether the dialog is a Warning dialog.
- **isQuestionDialog** Checks whether the dialog is a Question dialog.
- **isPlainDialog** Checks whether the dialog is a Plain dialog.
- **isOptionPane** Checks whether the dialog is an OptionPane dialog.

Next group helps to retrieve or check data from the dialog:

- **getTitle** Get dialog title text.
- **getOptionPaneMessageType** Get option pane type as string.
- **getMessage** Get dialog message text.

- **checkTitle** Check dialog title text against a given value.
- **checkMessage** Check dialog message text against a given value.

Then there is a group to check for availability and click specific buttons:

- **hasOkButton** Check for the OK button being available.
- **hasYesButton** Check for the Yes button being available.
- **hasNoButton** Check for the No button being available.
- **hasCancelButton** Check for the Cancel button being available.
- **clickOkButton** Click the OK button.
- **clickYesButton** Click the Yes button.
- **clickNoButton** Click the No button.
- **clickCancelButton** Click the Cancel button.

Finally there procedures to enable and disable a name resolver for option panes:

The need for a name resolver for this component results from the fact that neither the dialog nor its subcomponents, like confirmation buttons and message text labels, are named. So QF-Test needs to identify them on base of their titles or label texts, which results in creating various component nodes for each instance of JOptionPane. This may lead to a great number of those elements and is for reasons of clearness and maintainability not desirable.

Now let's have a look at the interface procedures. They are very simple:

- **enableNameResolver** Installs the JOptionPane name resolver.
- **disableNameResolver** De-installs the JOptionPane name resolver.

The procedure `enableNameResolver` installs and enables an extension to QF-Test called the JOptionPaneResolver. This name resolver provides convenient handling of JOptionPane dialogs. It is a standard `NameResolver` as explained in the technical reference manual.

All the other option pane procedures automatically activate (and deactivate) the name resolver. Only if you directly want to interact with option pane components you need to install the resolver yourself.

Then, it is typically sufficient to install the NameResolver once at the beginning of the test-run, e.g. directly after the Wait for client to connect node. De-installing commonly may not be necessary.

As mentioned above predefined generic components are provided within `qfs.qft` which dispenses with the need of maintaining your own ones. The following figure shows them:

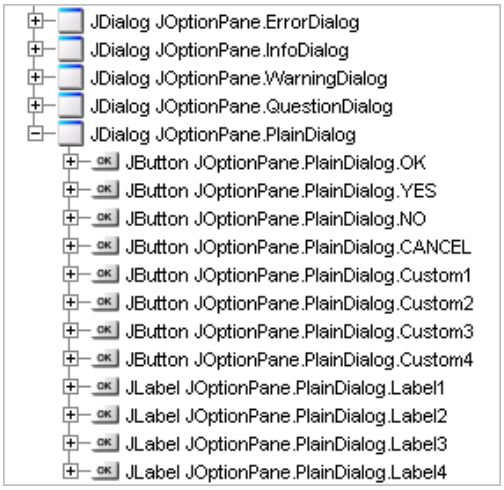

Figure 7.19: Generic OptionPane Components in `qfs.qft`

You can see from the figure, that there are 5 types of option pane dialogs supported:

- Error
- Info
- Warning
- Question
- Plain

As an example the plain dialog component node has been opened to show its content. There are predefined nodes inside for the different possible buttons (standard and custom buttons) and labels, usually containing the message text.

In our demo test-suite `StdLibDemo.qft` you can find a little test for this name resolver inside the ”OptionPane” test node which exercises the two ”About” Dialogs available in the ”Help” menu of the demo. You see both a test with and without the name resolver. Please have a look at the components under ”Windows and components” used for this test. Without the name resolver installed two slightly different components need to be maintained. When using the name resolver only one of the predefined components from `qfs.qft` is needed.

Note The implementation of the name resolver described in this chapter should be able to

map most common occurrences of JOptionPane components to the predefined components in `qfs.qft`. However, there may be instances of JOptionPanes with custom components inside, resulting in separate recording of component representations.

7.3.16 The SWT/FileDialog Package

The package `qfs.swt.filedialog` is intended to assist in manipulating any file dialog (the org.eclipse.swt.widgets.FileDialog SWT component).

Here is the list of available procedures within this package:

- **selectFile** Select a file within a file dialog window.

7.3.17 The SWT/ColorDialog Package

The package `qfs.swt.colordialog` is intended to assist in manipulating any color dialog (the org.eclipse.swt.widgets.ColorDialog SWT component).

Here is the list of available procedures within this package:

- **selectColor** Select a color within a color dialog window.

7.3.18 The SWT/DirectoryDialog Package

The package `qfs.swt.directorydialog` is similar to the FileDialog package. It is intended to assist in manipulating any directory dialog (the org.eclipse.swt.widgets.DirectoryDialog SWT component).

Here is the list of available procedures within this package:

- **selectDirectory** Select a directory within a directory dialog window.

7.3.19 The SWT/Instrument Package

The package `qfs.swt.instrument` provides procedures to instrument your Eclipse/SWT application correctly.

Here is the list of available procedures within this package:

- **setup** Instrument an Eclipse/RCP based or standalone SWT application, as described in the manual chapter on SWT instrumentation.

7.3.20 The AWT/Menu Package

The `qfs.awt.menu` package contains procedures to work with AWT menus. AWT menus are the only AWT components, which are not supported by QF-Test out of the box.

Here is the list of available procedures within this package:

- **selectItem** Select a given item of a menu.

7.3.21 The Run-log Package

The `qfs.run-log` package contains procedures, which writes specified messages into the run-log. This package has been introduced to give testers without scripting-knowledge the opportunity to write messages into the run-log.

Here is the list of available procedures within this package:

- **logError** Write a given error message into the run-log.
- **logWarning** Write a given warning message into the run-log.
- **logMessage** Write a given message into the run-log.

7.3.22 The Run-log.Screenshots Package

The `qfs.run-log.screenshots` package contains procedures, which write images into the run-log and some helper methods.

Here is the list of available procedures within this package:

- **getMonitorCount** Return the total number of monitors.
- **logScreenshot** Write a screenshot of the whole screen into the run-log.
- **logImageOfComponent** Write an image of a given component into the run-log.
- **logScreenshotOfMonitor** Write a screenshot of a given monitor into the run-log.

7.3.23 The Shellutils Package

The `qfs.shellutils` package contains procedures to support most common shell-commands.

Here is the list of available procedures within this package:

- **copy** Copy a given file or directory to a specified target.
- **deleteFile** Delete a given file.
- **exists** Check for existence of a given file or dirctory.
- **getBasename** Return only the file name of a full file name.
- **getParentdirectory** Return only the directory name of the full file name.
- **mkdir** Create a given directory. It also creates non-existing directories in path.
- **move** Move a file of directory.
- **touch** Create a specified file.
- **removeDirectory** Remove a specified directory.

7.3.24 The Utils Package

The `qfs.utils` package contains procedures, which covers common helper-functionality during test-development.

Here is the list of available procedures within this package:

- **getDate** Return a string containing the date. Default is the current date. (Other dates can be configured.)
- **getTime** Return a string containing the time. Default is the current time. (Other timestamps can be configured.)
- **logMemory** Log current memory use.
- **printVariable** Print the content of a given variable to the console.
- **printMessage** Print a given message to the console.
- **writeMessageIntoFile** Write a given string into a given file.

7.3.25 The Database Package

The `qfs.database` package contains procedures to execute SQL commands on a database.

Please note, that the class of the database-driver must be in QF-Test's plugin directory or in the CLASSPATH before QF-Test startup.

To get more information about the connection-mechanism to your database, please ask your developers or see www.connectionstrings.com.

Here is the list of available procedures within this package:

- **executeSelectStatement** Execute a given SQL-Select-Statement. It stores the result in a global variable ”resultRows” on the Jython variable stack.
- **executeStatement** Execute a given SQL-command. Here any SQL command can be specified.

7.3.26 The Check Package

The `qfs.check` package contains procedures to do checks.

Here is the list of available procedures within this package:

- **checkEnabledStatus** Check, whether a component is enabled or disabled. It writes an error into the run-log, if failing.
- **checkSelectedStatus** Check, whether a component is selected or not. It writes an error into the run-log, if failing.
- **checkText** Check the text of a component. It writes an error into the run-log, if failing.

7.3.27 The Databinder Package

The `qfs.databinder` package contains procedures for execution within a ”Data driver” node which bind data for iteration.

Here is the list of available procedures within this package:

- **bindList** Create and register a databinder that binds a list of values to a variable. Variables are separated by whitespace or by a given separator character.
- **bindSets** Create and register a databinder that binds a list of value-sets to a set of variables. Value-sets are separated by linebreaks. Variables within a value-set are separated by whitespace or by a given separator character.

7.3.28 The Web Package

The `qfs.web` package contains procedures to support web-testing.

Here is the list of available procedures within this package:

- **startBrowser** Launch an SUT client with a browser.

Chapter 8

Managing Complex GUI Components [30-45 min]

Up till now, we've dealt with relatively straight-forward java GUI components - such as radio buttons or text fields - which we refer to as "simple" GUI elements. We now take on more complex components which themselves contain sub-elements. The classic examples for such complex components are trees and tables.

Take the nodes of a tree. A node is not itself a GUI component in the Java sense, it is but a graphical representation of some data. This technical distinction is not much help for testing, where the behavior of such a tree node is just as important as the behavior of a simpler GUI element.

To handle these cases QF-Test uses a special element called an "item" that represents complex sub-components. In this section, we'll deal with the manipulation of items and the syntax used by QF-Test to refer to them.

8.1 The SUT For Testing

For this section we'll use a simple application as an SUT that contains several complex elements. We've called it "ItemsDemo" and it resides within the distribution Jar-file included as part of your current QF-Test version.

You can create the setup node yourself as shown in the figure below or open the `items.qft` test-suite from the tutorial directory that contains a respective setup sequence within the "Extras" node.

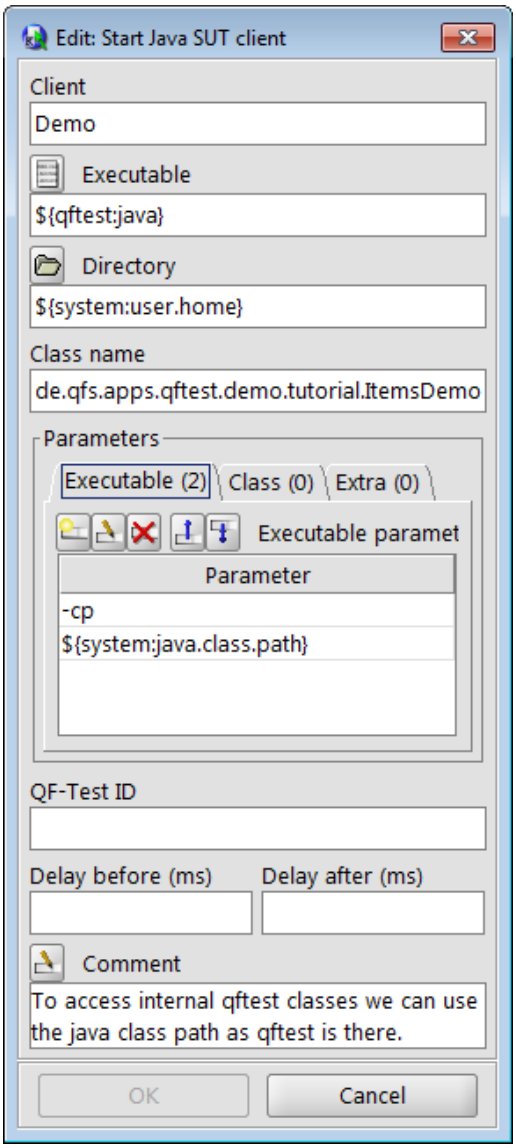

Figure 8.1: Start Java SUT Client Node for ItemsDemo

Start the SUT. You should see the following:

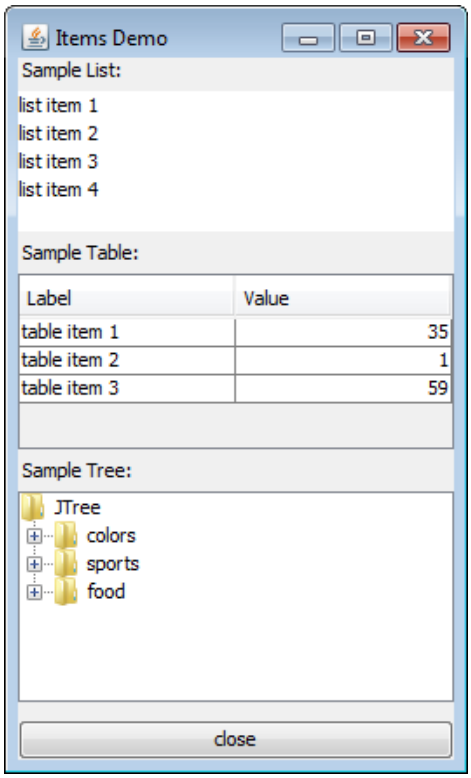

Figure 8.2: The ItemsDemo Window

8.2 One-Dimensional Complex Components

We'll start by analysis of the simplest of complex elements, which is a one-dimensional list such as represented by the JList Java class. Within the SUT window you'll see a JList as the first component, labeled as "sample list."

Start recording and perform a few clicks on the elements of the sample list. When you stop recording, you'll see a sequence such as this:

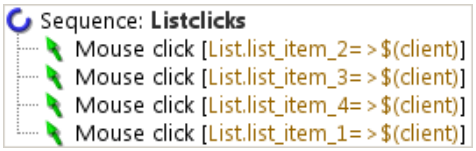

Figure 8.3: Mouse Operations Performed on a JList

Click on one of the recorded "Mouse click" nodes and look at the component that is being referenced.

Figure 8.4: Reference to a JList Item

The reference is fairly straight-forward with a ”parent.child” type of structure. In this case ”List” is the name of the parent component (the JList), and ”list_item_2” is the name the individual (child) item of the list. Let’s look at how this component is actually stored by QF-Test though.

With the same node still selected, run ”locate component” by either right-clicking on the node and clicking on the ”locate component” option, or simply use the Control-W shortcut key. You should now see an expanded list of the registered elements of the SUT in the ”Windows and components” section of the test-suite, with the node for the list item selected, as such:

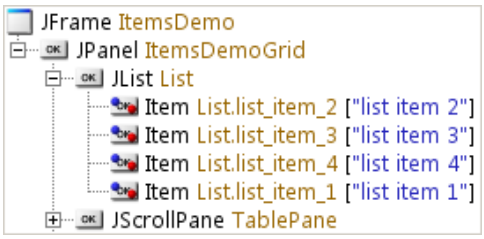

Figure 8.5: JList Item Node

What we are seeing here is a node-type called an ”Item” that is used specifically for the purpose of describing elements of complex structures like our list. An Item node consists of two main components: the parent it belongs to and the index (or indices) of the item(s) within the parent. Look at the properties for this Item node:

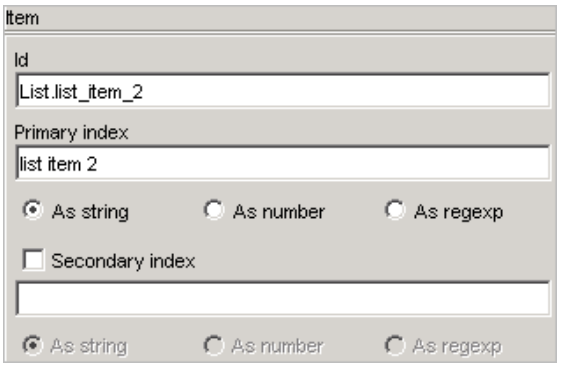

Figure 8.6: JList Item Properties

Here we see the QF-Test ID of the component (with the QF-Test ID of the parent component prepended) and a primary index. Since the list is a one-dimensional structure, we only need one index for now. When we look at tables in the next section, the secondary index will come into play.

The primary index is described in this implementation as a string, which is just the actual text of the item in the SUT's list. This type of implementation may be fine for most applications, but of course will have bad consequences if the text of items in the list ever changes. To avoid this problem, QF-Test presents you with two other possibilities, the first of which is referencing the item as a numerical index (starting with index 0) of the list. Please change the index "list item 2" to the equivalent numeric index "1". Then change interpretation of the primary index by clicking on the "as number" radio button:

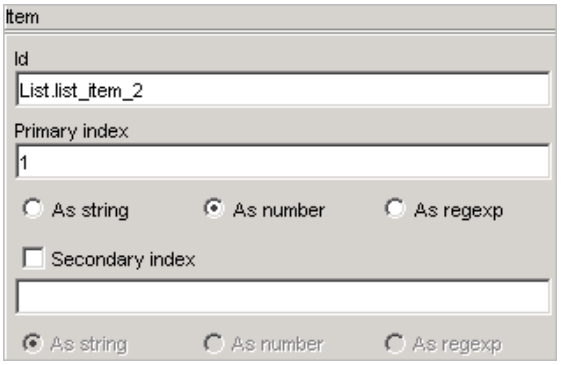

Figure 8.7: JList Item with Numeric Index

Another possibility for indexing of items is given through the sophisticated regular-expression index. QF-Test will perform a textual search of the items in the list until a pattern matching the regular expression you supply is found. For example:

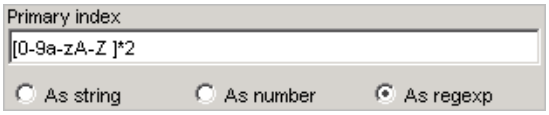

Figure 8.8: JList Item with Regular Expression Index

The regular expression `[0-9a-zA-Z ]*2` given here indicates an arbitrary number of alpha-numeric characters or white spaces and with the digit "2" required to be at the end. Another option would be the expression `.*2`. For more information about writing regular expressions, please consult the "Technical details about miscellaneous issues" chapter of the technical reference manual.

Try executing your recorded sequence for the list again, but now with the modified Item

nodes. As long as you modified the index references correctly, the behavior of the sequence should be exactly the same.

8.3 Two-Dimensional Complex Components

We'll now move to a more complex structure: a two-dimensional component best represented by the Java JTable class. A table, of course, consists of rows and columns. So an item within a table requires two indices in order to be properly referenced.

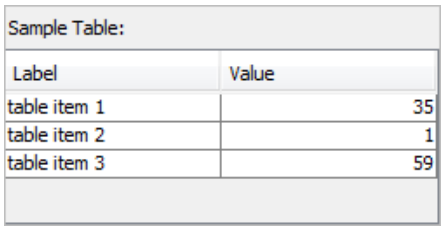

Figure 8.9: The Sample Table within the SUT

As you did with the list, try recording a few clicks now on items within the "sample table" of the SUT. Make sure to record clicks on elements of both columns. When done, you'll have something like this:

Figure 8.10: Mouse Clicks Recorded from the JTable

The nodes should look essentially similar to the nodes recorded from the JList structure. Here we have simply an "Mouse click" node that references a component in the form of "parent.primaryIndex&secondaryIndex". At this "Table" is the JTable parent, "Label" or "Value" is the primary (or column) index and a number represents the secondary (or row) index. The "&" before the secondary index has a particular meaning. It indicates that the secondary index directly refers to the given table item and there is no Item node used for the access. This special syntax is described in detail in the next chapter. Using "locate component" take a look at the actual Item nodes that were registered:

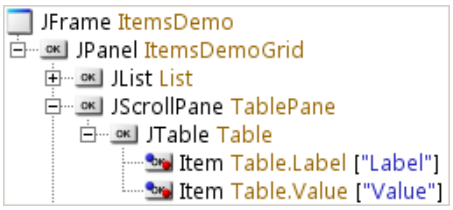

Figure 8.11: Registered Items Nodes for the JTable

Because Item nodes are not used for accessing the secondary index, such were only recorded for the primary index, i.e. for the columns ”Label” and ”Value”. Taking a look at the details of the ”Label” Item node, we see that only the primary index is used inside the Item node.

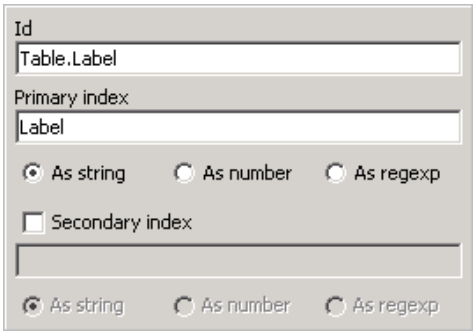

Figure 8.12: Properties of a JTable Column Item Node

8.4 Item Nodes versus Syntax

The previous chapter explained the representation of items within QF-Test. But there is another way to access elements of complex components. This is done by referring the target element by its parent component extended with a certain syntax specifying the subcomponent. This type of syntax is what QF-Test calls a ’direct access’ to the item. Equivalent to the item representation there a three possible syntax types:

- **@** - textual
- **&** - numeric
- **%** - regular expression

The secondary index in the mouse clicks shown in figure 8.10(113) is represented by this direct numeric syntax using the ”&”. No item node is needed for in this case. Please try to understand the different representations.

Whether sub-items are represented as item nodes or as syntax doesn't make any difference from QF-Test's point of view. Both methods have their merits. Item nodes are well suited to static elements whereas syntax is almost always preferable if there's a certain degree of variability. QF-Test tries to record sub-items as intelligently as possible, but in some cases it may be preferable to override its defaults. Using the options for sub-item recording (see manual) you can define precisely how QF-Test should record sub-items.

As concluding example, the following figure shows a "Mouse click" sequence where also the primary index is represented as syntax. The functionality remains the same.

Figure 8.13: Mouse Clicks using the syntax for direct access

8.5 The Importance of Uniqueness

We pause here a moment to reiterate the importance of uniquely referencing items within a complex component. As the nature of the component increases in complexity, so does the complexity of referencing the items of the component increase. For the previous two examples (lists and tables), the structure is fairly simple. But even with these components, duplicate entries could appear. Consider the following table:

Expenses and Income per Year:

Year	Expenses	Income
2001	$5061	$8500
2002	$4985	$9400
2003	$6200	$9400

Figure 8.14: A Table with Duplicate Items

Since duplicate items can (and do) appear, you need to consider carefully how your test-suite would handle such a table. What could change, and will remain static? The answers determine what type of reference should be made for items within the table. This is why QF-Test uses the row number as secondary index when recoding components in intelligent mode. The mode that specifies the way sub-items are recorded in QF-Test can be adapted via Edit→Options... and then select "Record" -> "Sub-items".

In the next section we'll tackle the most complex of components, in which handling of potential duplicate items is of critical importance.

8.6 Trees

Trees are a special case of complex components, since their hierarchic framework does not map well to a linear structure. The uniqueness of a node must also be taken into special consideration.

However, given the set of tools and concepts learned in the previous sections, QF-Test makes manipulation of trees very workable. Of special importance is the introduction of the '/' operator, which works just like the path-separator of a file-system hierarchy.

Let's first just give it a try. As before, record a few clicks on the "sample tree" of the SUT. You should end up with something similar to this:

Figure 8.15: Mouse Clicks Recorded from the JTree

As a first step, try just adjusting and playing with your recorded sequence to understand how trees are handled.

Clicks at +/- symbol for expanding and collapsing nodes are recorded with negative X coordinate as the clicks needs to be places left of the node. For playback this kind of clicks are not needed as QF-Test will automatically expand the tree for you, provided that the respective option is set.

Now let's look at the syntax, such as with the click on the node "/JTree/colors/yellow". The path operator ('/') is put to use by describing the route taken down the tree to find the proper node. This type of scheme has tremendous advantages in that immediately we eliminate the uniqueness problem of another node with the same name located in a different branch of the tree! The only uniqueness problem that remains is if there are tree nodes of the same name located within the same branch (i.e., there are two "yellow" nodes located in "/JTree/colors"). In this case, we can revert back to our trusted mechanism of referencing a node numerically.

Modification of the indexing of a JTree node is best accomplished using the direct access method that was introduced in the previous section. Here, we use one of the

operators '@', '&', or '%' to (respectively) allow direct textual, numeric or regular expression reference. Here's how a numeric reference to the same node would look using direct access:

Figure 8.16: Direct Numeric Reference to a JTree Node

Easy enough! You should have the hang of it now to traverse your way through the complexities of such components.

Chapter 9

It's time to start your own Application

After having spent a lot of time with all those example programs, you are ready now to start up your own application (if you really haven't done that already).

The Quickstart Wizard that is available via menu Extras→Quickstart Wizard... helps you to achieve this. Simply follow the wizard steps to generate a proper setup sequence. Please refer also to chapter 3 "Quickstart" in the user manual.

Then go on utilizing the lessons you've learned in this tutorial - record small sequences of events and checks, create procedures etc. to build up your testing library.

Finally we reached the end of the basic tutorial part.

Part II

Web testing with QF-Test

Chapter 10

Working with a Sample Test-suite [30-45 min]

10.1 Starting QF-Test and Loading the Test-suite

After starting up QF-Test, you can immediately bring up our first example test-suite. You can locate the test-suite file by bringing up the open-file dialog window with the File→Open menu option. Select the directory `qftest-4.0.8/doc/tutorial/web` from your QF-Test installation. From this directory, select the file `webdemo.qft`. QF-Test will then load the indicated test-suite which should look as follows:

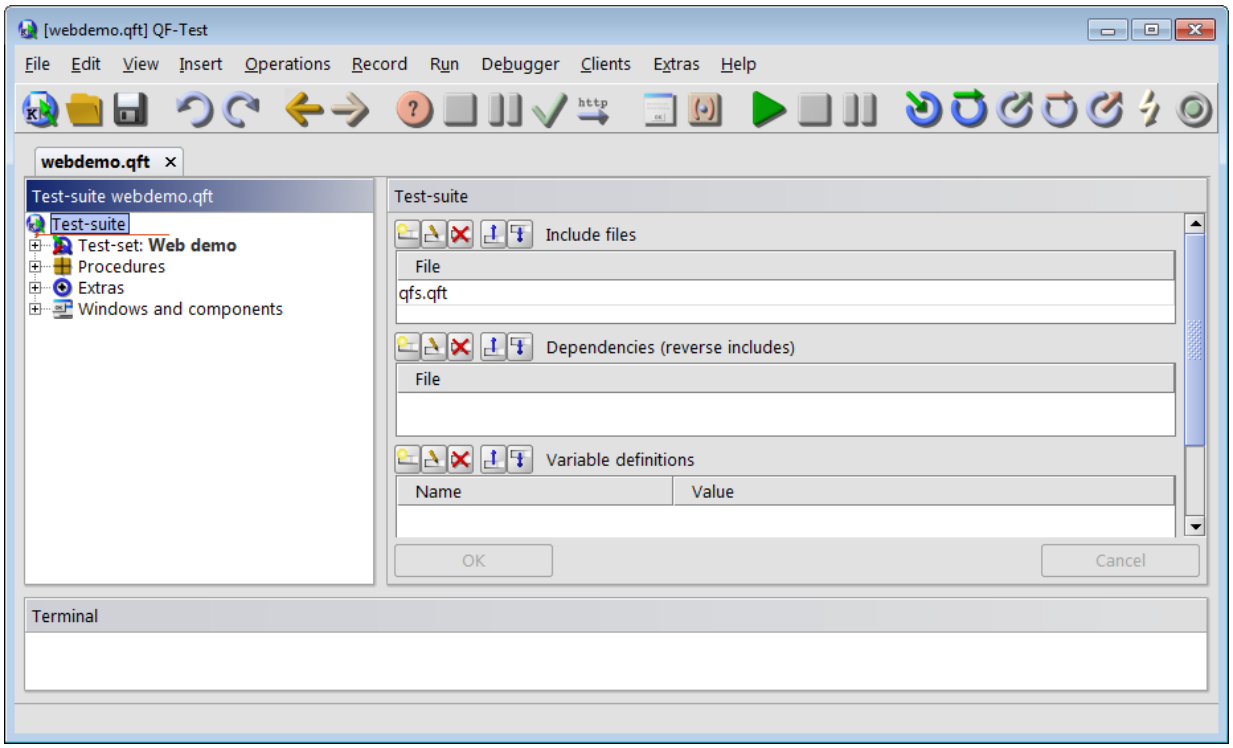

Figure 10.1: The Test-suite `webdemo.qft`

The left part of the main window contains the tree structure that represents the test-suite. The right side displays the details of a selected node. (In case you don't see the details view, you can switch it on with the View→Details menu option.) At the bottom of the main window you'll see the terminal output area, which displays standard messages and communications in between your test-suite and the client application you are testing. Displaying the terminal output can be toggled on or off using the View→Terminal→Show menu option.

With the help of the tree structure in the main window, you can navigate and select individual nodes of the test-suite. When a node is selected, you'll see its properties displayed in the details area on the right.

The top level of the tree consists of following node types:

- "Test-suite" - This is the root node of every test-suite. All other node are subordinated to it.
- "Test-set" - This node type allows collecting and structuring of test-cases.
- "Procedures" - This node is meant to contain the procedures of the test-suite.

- ”Extras” - This node serves as a playground for experimenting and assembling of test-cases. Here you can insert arbitrary nodes independent of their usual restrictions.
- ”Windows and Components” - At this position QF-Test collects the information on the elements and structure of the document to be tested. Those is gathered automatically during recording and necessary for finding the proper components when replaying the tests.

Click on the node ”Test-set: Web demo” to expand the nodes underneath it. You’ll see that basically three test-cases are contained in this test-set: ”Clickstream”, ”Text check” and ”Selected test”. Those test-case nodes are enclosed by a ”Setup”/”Cleanup” pair to ensure a proper system as base to run the test-cases on.

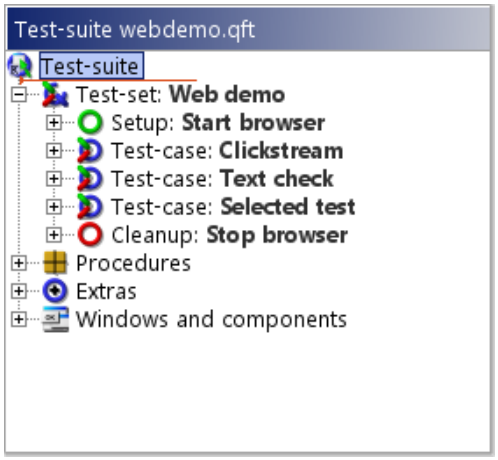

Figure 10.2: The Content of the Node ”Test-set: Web demo”

In the following sections we’ll describe the purpose and function of these individual nodes.

10.2 Starting the browser

Our first step will be to examine the ”Setup” node. Expand this node now, so that you see the contents as shown in the following diagram:

Setup: Start browser
Start browser: ie - file://${qftest:suite.dir}/${system:user.language}/index.html [web]
Wait for client to connect [web]
Wait for component [index=>web]

Figure 10.3: The ”Setup: Start browser”

Within the ”Setup” node you’ll see three child-nodes:

- ”Start browser” - starts the browser to be used for web testing.
- ”Wait for client to connect” - waits until the browser finishes its startup and connects itself to QF-Test.
- ”Wait for component” - waits for the appearance of a certain component of the web page, in our case the man document `index.html`. Don’t worry if this step isn’t completely clear at the moment, we’ll explain later about components and their meaning.

Now click on the ”Start browser” node so that it is selected as seen below:

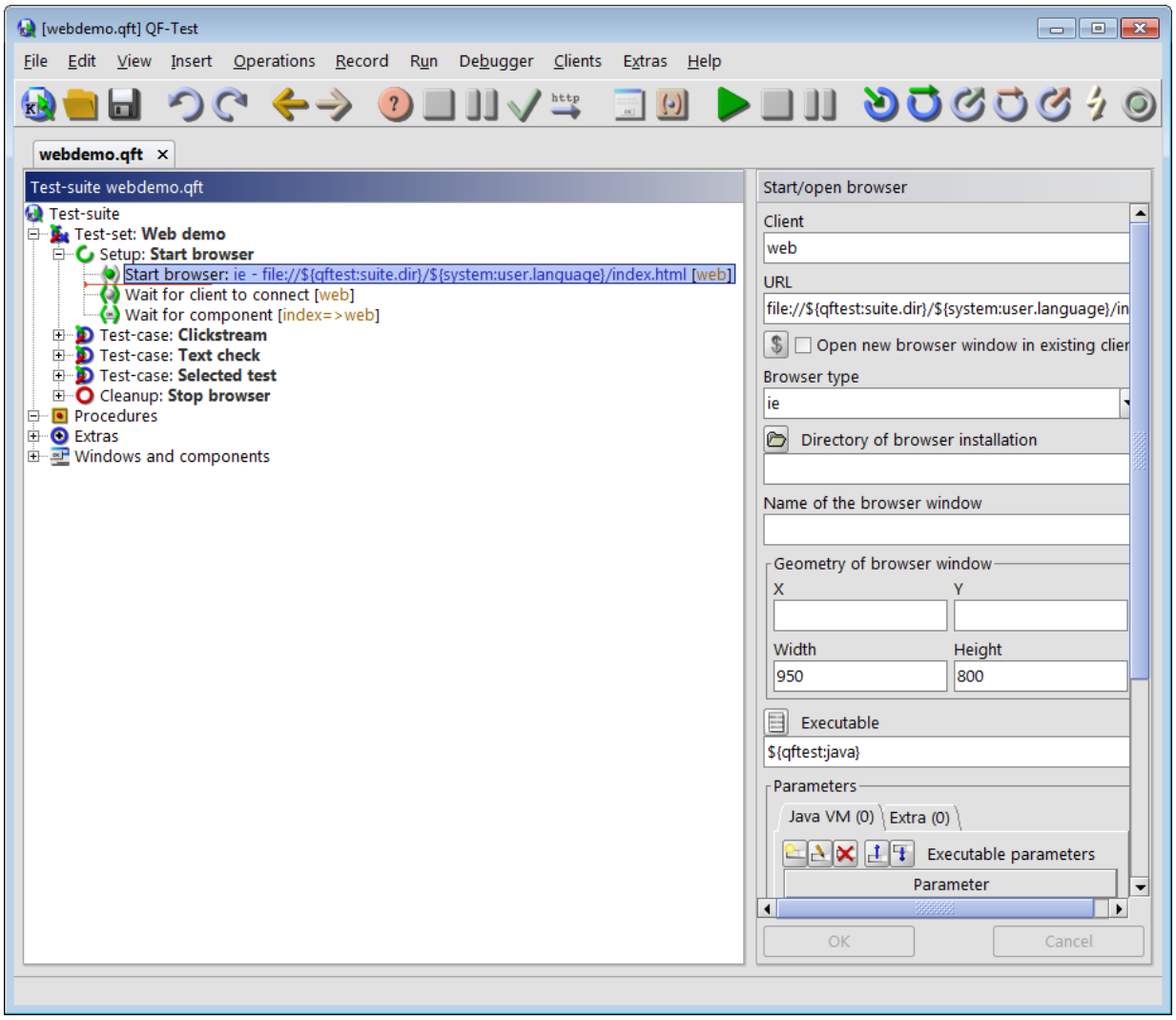

Figure 10.4: The ”Start brower” Node

Let’s take a closer look at the properties of this node, as displayed in the details window on the right side:

- In the first field, labeled ”Client,” is an unambiguous name for the SUT application. You can give the SUT any name you wish, but the same name will be used as a reference to the SUT in other nodes. In other words, this name must be used consistently.
- The ”URL” is the second field, in which the full descriptor of the web page is specified where testing shall start. As we use a local site in the file system, the descriptor starts with `file://`. It is followed by two variable parts: `${qftest:suite.dir}` is expanded to the directory of the current test-suite.

`${system:user.language}` will result in `en` or `de` dependent on the user language setting. Finally there is the website `index.html`.

- Also important is the fourth field, which contains the type of browser to be used for testing. ”ie” means Internet Explorer. The alternative is ”mozilla” for Mozilla Firefox or ”chrome” for Google Chrome. If you choose ”mozilla”, you must also specify the location of your browser installation in the field ”Directory of browser installation”. The other browsers are located automatically and that field is ignored.
- The width and height entries in ”Geometry of browser window” ensure the web page is fully visible. However this is more for cosmetic reasons and has no result on the test functionality.
- The remaining property fields can be left empty and are not required for our example. You will note, however, that a comment has been filled in (the last field). All nodes have an optional comment field which you as a test-suite developer may find helpful for documenting a node’s purpose.

If you like, you can take a look at the details of the other two nodes inside the setup sequence as well.

At this point, we’re ready to start the SUT. Click on the ”Setup” node again (so that it is selected), but still expanded (the child-nodes still visible). Now click on the ”Start test run” button . This button causes the selected node - ”Setup” in our case - to be executed.

When a node executes, QF-Test will use the tree structure to show you what is happening. An active node will be marked with an arrow pointer ”->” that indicates which node is being executed. You’ll also see messages from the SUT client in the terminal display.

When the ”Setup” sequence is completed, the browser window showing the demo page will appear on your screen. This is the SUT, fully under the control of the watchful eye of QF-Test.

Note If the browser window is not visible on your screen or only appears for a short moment, it is probably covered by the main window of QF-Test. The best solution is to place both windows side by side so you can watch them at the same time.

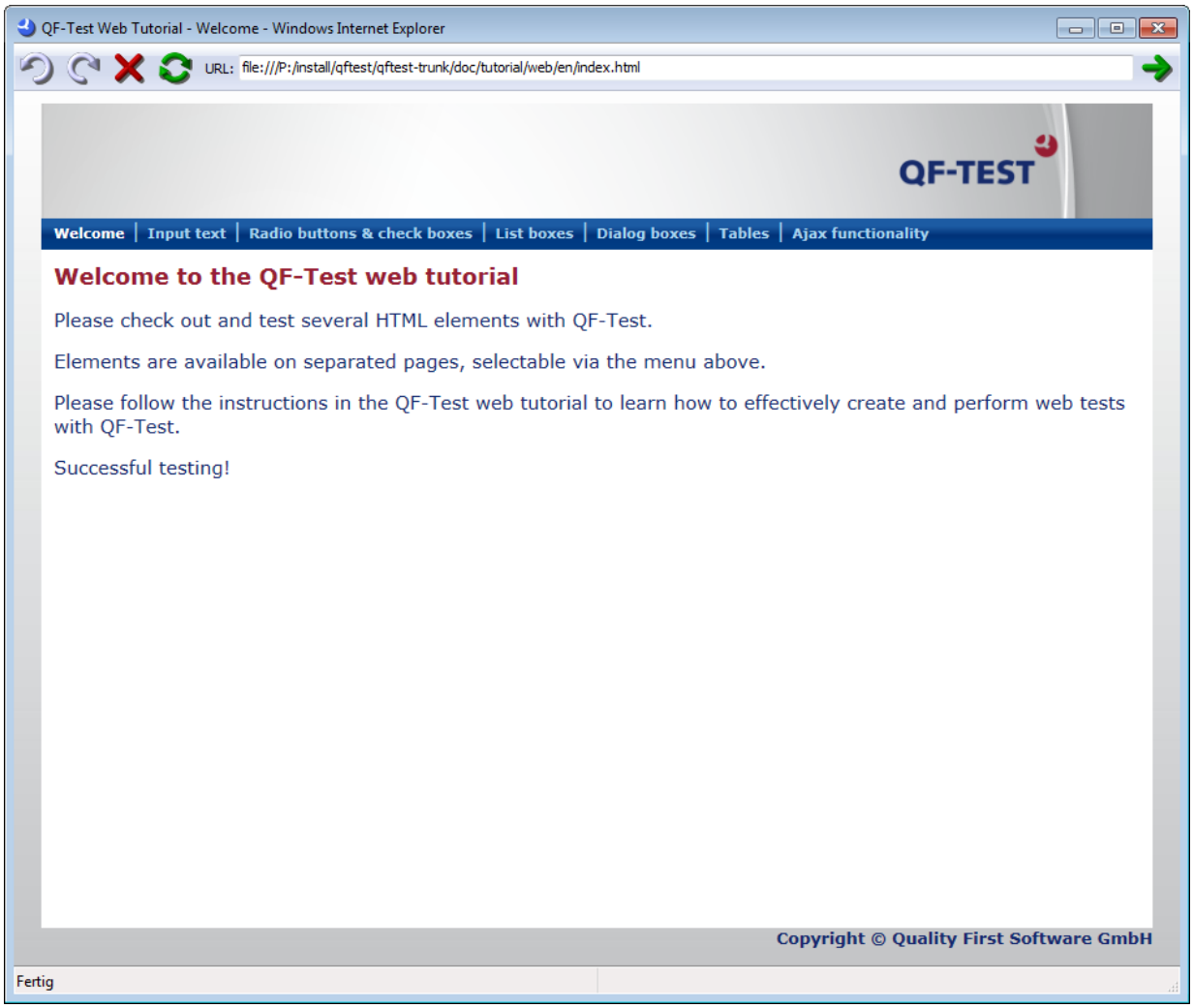
QF-Test Web Tutorial - Welcome - Windows Internet Explorer
URL: file:///P:/install/qftest/qftest-trunk/doc/tutorial/web/en/index.html
QF-TEST
Welcome | Input text | Radio buttons & check boxes | List boxes | Dialog boxes | Tables | Ajax functionality
Welcome to the QF-Test web tutorial
Please check out and test several HTML elements with QF-Test.
Elements are available on separated pages, selectable via the menu above.
Please follow the instructions in the QF-Test web tutorial to learn how to effectively create and perform web tests with QF-Test.
Successful testing!
Copyright © Quality First Software GmbH
Fertig

Figure 10.5: The Web Demo Page in the Browser Window

10.3 Clickstream Test-case

Our next task is to take a first look at how a test-case for the SUT in QF-Test is structured. The test-case node labeled ”Clickstream” contains a sequence of mouse-clicks to all menu items in the web site menu.

When you expand the ”Clickstream” test-case node, you’ll see the sequence as shown below:

Figure 10.6: The Clickstream Test-case Node

Before executing the test please ensure the ”Welcome” page is selected within the web demo. Then mark the ”Clickstream” test-case node and click the replay button . The sequence is supposed to pass through without problems.

The test result is indicated during and after the test run in the status line at the bottom of the QF-Test main window and says ”No errors”. Next to it there are counters for the number and results of the executed test-cases. In our case it was just one that performed error-free which means a success rate of 100%.

Hint: Let the mouse cursor hover above a counter icon to see its description. A listing of all possible counters can be found in chapter Capture and replay of the manual.

For a better understanding of what you saw during replay, let’s take a look at the sequence ”Menu items” within the ”Clickstream” test-case:

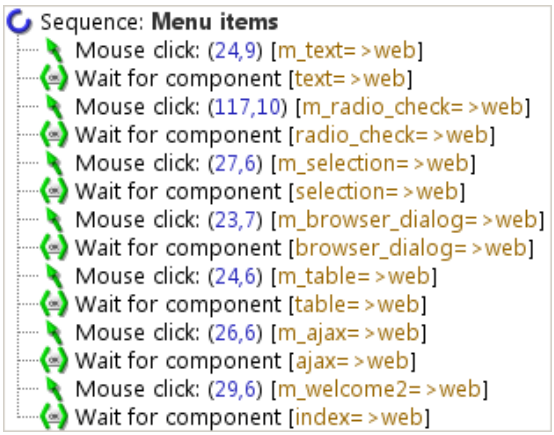

Figure 10.7: The ”Menu items” Test Sequence

Here we see that the sequence starts with a mouse click. Next to the ”Mouse click” label you’ll see the actual coordinates used to implement the mouse click relative to the component listed in the brackets next to the coordinates. The first mouse click node is, for example, a click on the second menu item ”Text input”.

After the first mouse click there is a ”Wait for component” node that causes the execution of the sequence to wait for the appearance of the new web page. After the page is loaded successfully, execution of the sequence will continue with the next mouse click. It works the same way for selecting all remaining menu items until we are back at the ”Welcome” page.

10.4 A Few Tips

We take a short break in this section to give a few tips that might prove helpful once you continue on with the tutorial.

While working with various nodes in QF-Test you may, for example, require some assistance in remembering what purpose they are used for. To serve this need, QF-Test comes with context-sensitive help, driven simply by your mouse. Move the mouse pointer over an element you would like to have assistance with, then click the right mouse-button. In the now appearing popup menu, you'll see an entry labeled "What's this?" By selecting this option from the popup menu, the appropriate section in the user's manual will be brought up and displayed in your standard browser. You'll hopefully find the manual quite helpful and complete in answering questions that may arise.

We should also mention that there is a PDF Version of the user's manual available in the installation directory of QF-Test. Within the PDF version you may find it easier (in comparison to the HTML version) to search for specific terms and concepts.

10.5 Text Check

One of the most important concepts in QF-Test is that of a check; that is, a query of certain elements in the SUT. A 'text check' queries the existence or absence of text within a component of the SUT, such as a text field.

For example, let's make a check of a field in the SUT client. In our web demo page please select the second entry in the menu bar which is "Input text". Within the respective page you'll see a text field labeled "Name:" For our example, we'll do a check of this label text.

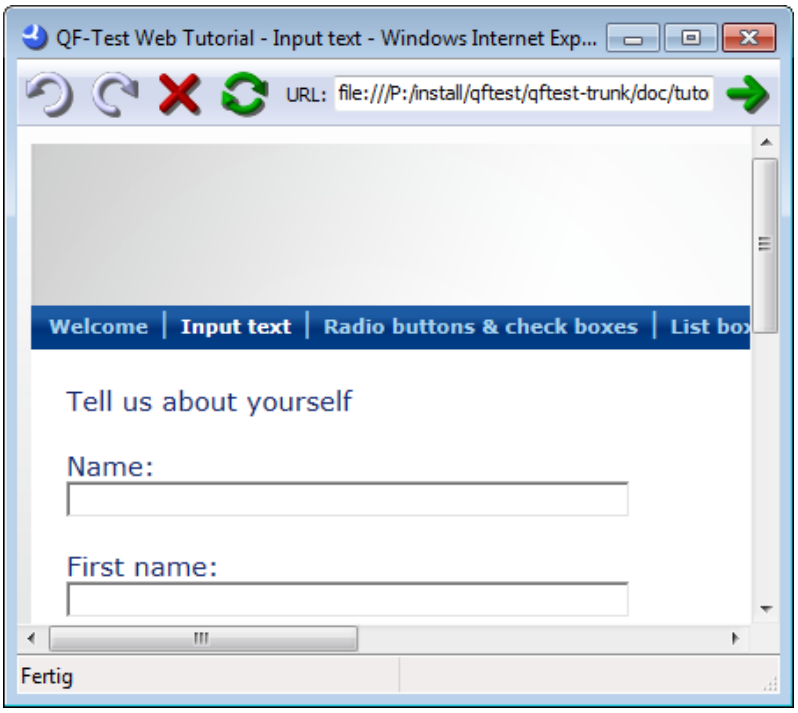
QF-Test Web Tutorial - Input text - Windows Internet Exp...
URL: file:///P:/install/qftest/qftest-trunk/doc/tuto
Welcome | Input text | Radio buttons & check boxes | List box
Tell us about yourself
Name:
First name:
Fertig

Figure 10.8: The ”Name:” Label to Check

Now return to your test-suite and expand the ”Text check” test-case node. Within you will see the following test sequence:

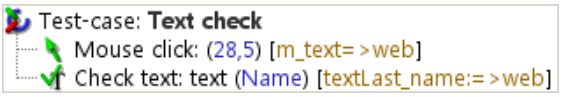

Figure 10.9: The Text Check Test-case

It consists of one click, aiming to open the ”Input text” page and a ”Check text” node for the label text.

First return to the ”Welcome” page within the web demo. Then with the ”Text check” test-case node selected, click on the replay button to execute it. When the sequence completes, a dialog-box will appear indicating that one error and one warning occurred:

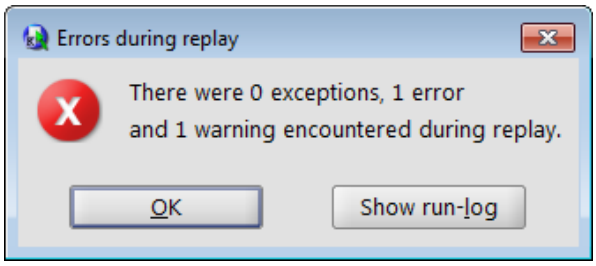

Figure 10.10: Error in String Check Node

What happened? Whenever such incidents occur, taking a look at the QF-Test run-log can prove very useful for diagnosis. You can open it directly via the "Open run-log" button on the error dialog. Alternatively the latest run-log can be opened from the test-suite via the Run→1. ... menu item or by pressing Control-L. A new window will then appear displaying the recorded contents of the sequence you just executed:

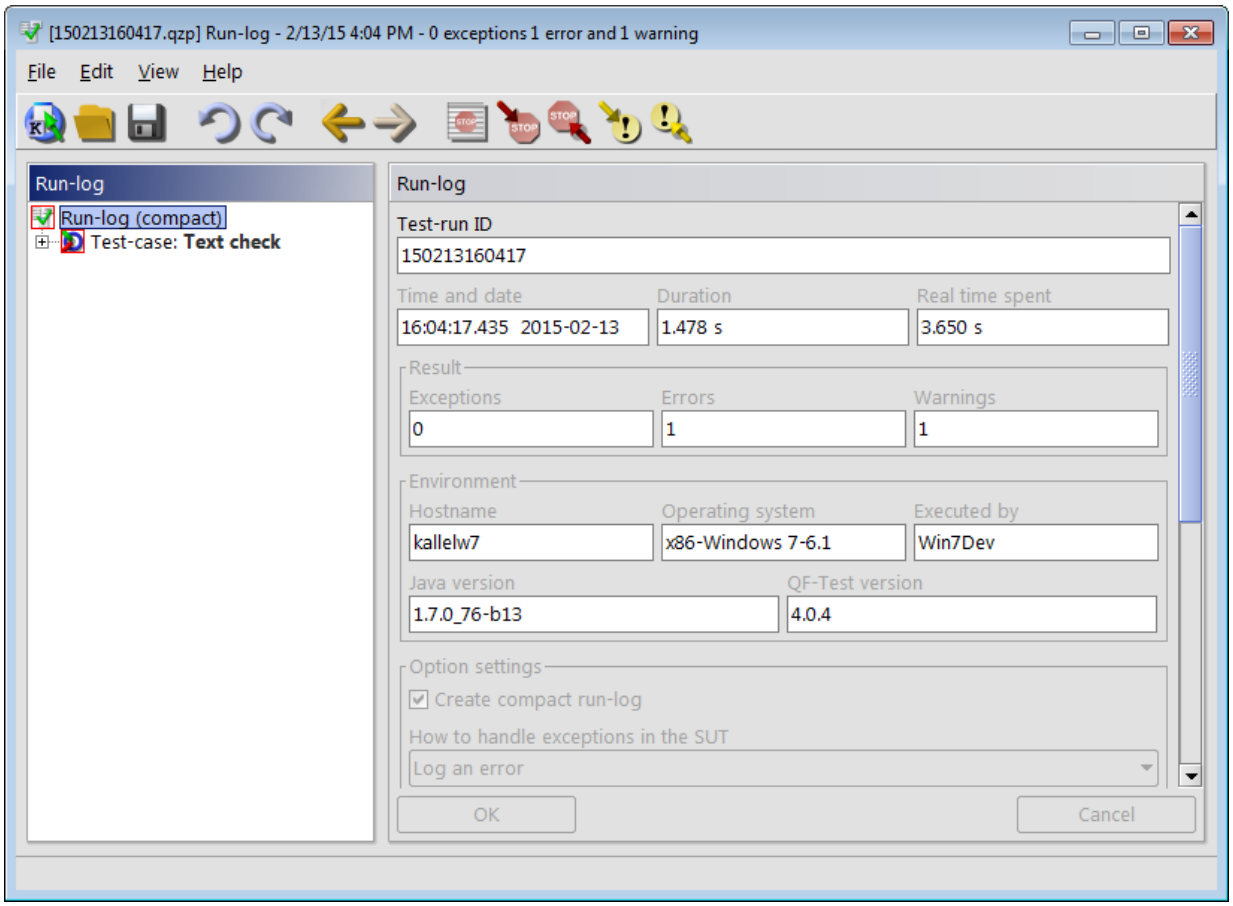

Figure 10.11: Run-Log for the Text Check Sequence

The run-log is similar in structure to the tree structure you are already familiar with from the normal test-suite view. The tree on the left side represents the time-recorded events of the test run, with the first event at the top, the last at the bottom. When you click on one of the event nodes on the left side, the properties of the event will be displayed on the right.

As you expand the nodes in the tree structure on the left, you'll notice immediately that some nodes are surrounded by a suspicious-looking red box. As you may suspect, this is an indication by QF-Test locating where a problem occurred in a child-node. If you keep expanding the red nodes, you'll eventually come to the actual problem node. An easier way to find the error is to simply use the Edit→Find next error menu option of the run-log's menu. This leads to the following:

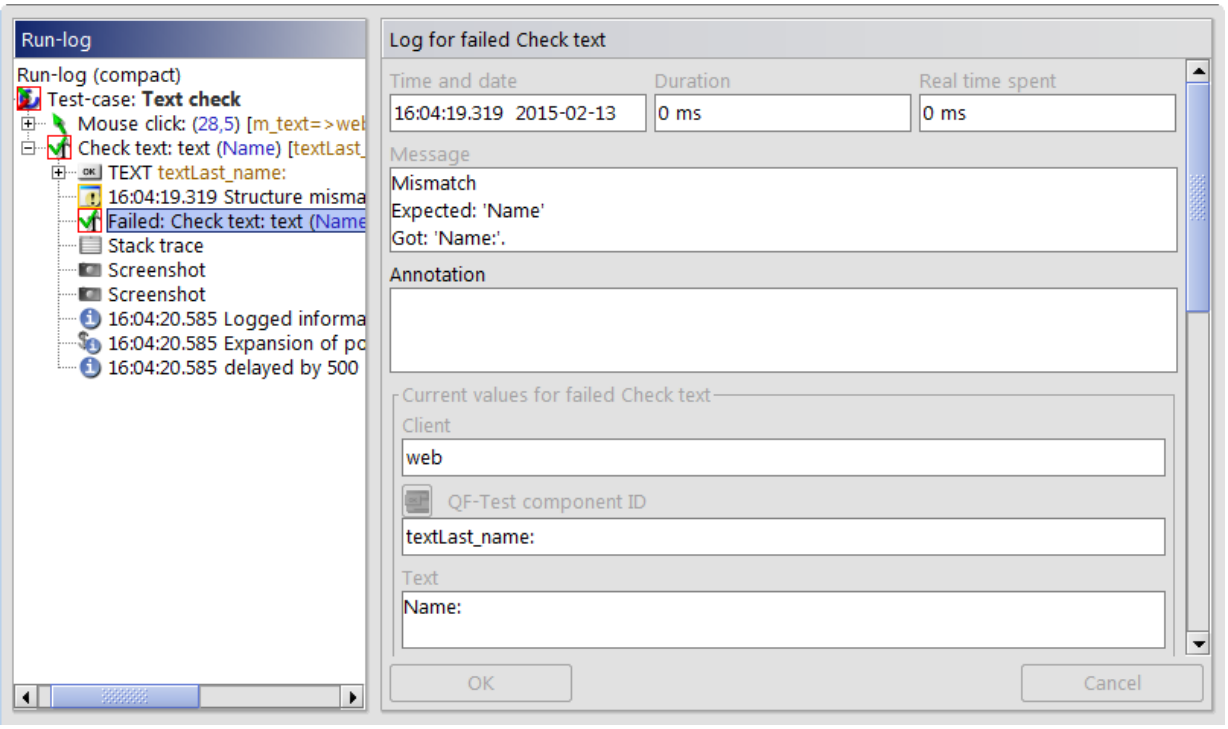

Figure 10.12: Diagnosis of Error in the Text Check Sequence.

When you click on the final red node, you'll see in the details of this node the deviation between the expected and found label text. The difference is actually a missing ":" at the end. Of course we have intentionally introduced this error into the test for demonstration.

The typical next step is to jump to the corresponding node in the test-suite and correct it. Provided that the appropriate node in the run-log is selected, you can use Edit→Find node in test-suite or simply type Control-T. One step beyond goes a functionality provided by the run-log node titled "Failed: Check text" in figure

10.12(131) available from the context menu, to be opened via a right click - Update check node with current data or Control-U.

Another useful feature for error diagnosis is the tree node labeled ”screenshot” in the run-log. Its details contain a full screenshot taken at the time when the error happened. Being able to see the state of the SUT at that moment is a valuable aid in determining the cause of the error. The following image shows a screenshot node:

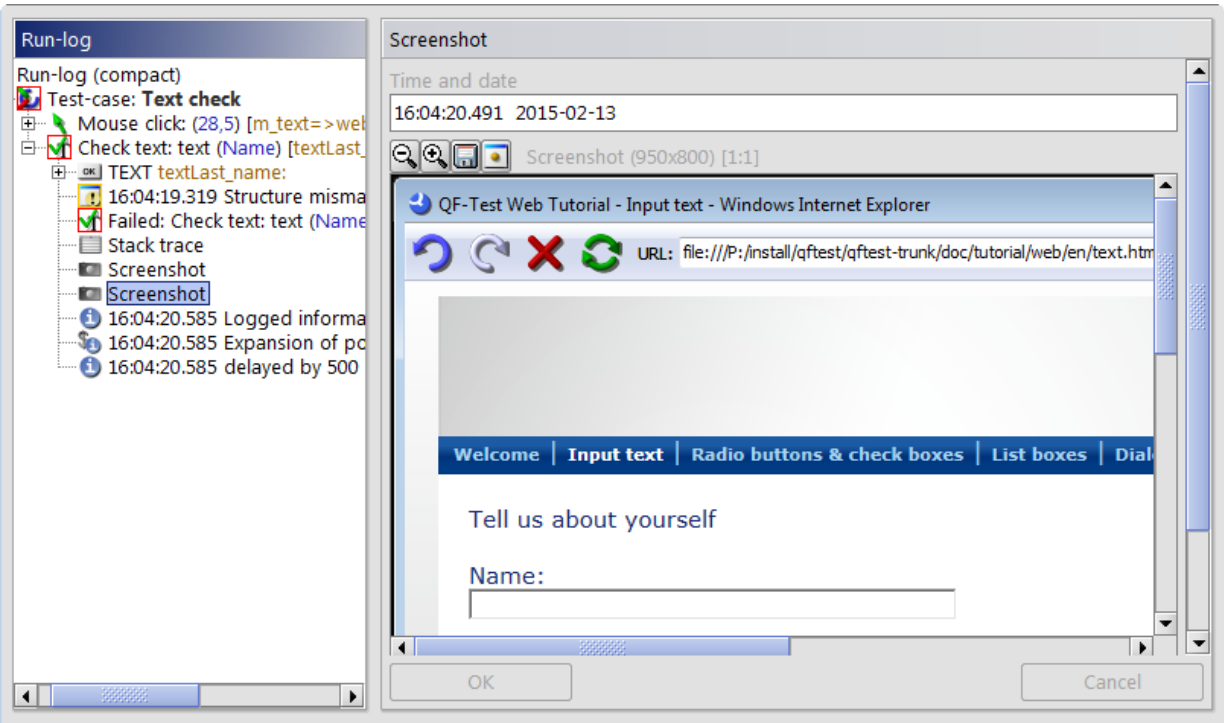

Figure 10.13: Screenshot node showing the error situation

Beside a screenshot of the whole screen, QF-Test also saves independent images of all SUT windows. For our example it is the protocol node ”Screenshot of window: Web demo”. Such can be helpful when the SUT is hidden by another application window at the time when the error occurs.

Note

The information gathered from a long test-run accumulates and can devour enormous amounts of memory. Therefore QF-Test is typically configured to create a special form of compact run-log. Herein only the last 100 run-log nodes are kept, the rest is discarded, except for information relevant for report generation and for error diagnosis which is always retained. This functionality is configurable through the option ”Create compact run-log” within Edit→Options →Run-logs→Content. The type of a run-log is also shown in its root node. The number of screenshots stored in the run-log can be configured as well.

Now a remark about the warning which occurred beside the error during the test-run. In the run-log of figure 10.12(131) it is marked by a yellow frame around the respective node. It's the one just above the erroneous check text node. The warning says: "Component 'textLast_name:' has no name". For the moment we don't want to track this warning any further. But naming of components is an important topic concerning test stability.

As a simple exercise, you may wish to modify the expected text in the "String check" node of your test-suite. Now when you execute the sequence, no error will occur. Please reverse the text to its previous state by Edit→Undo or Control-Z once the test has finished, because we will need the error for demonstration purpose later on.

10.6 Checking a Radio Button

The third test sequence "Selected test" performs queries about the state of a specific radio button in the SUT in order to determine if that radio button is selected. On the "Radio buttons & check boxes" page you can interactively select an number radio buttons. We'll use these radio buttons to implement a test that includes a successful check as well as a check that leads to an error.

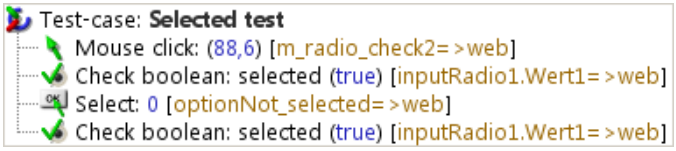
Test-case: Selected test
Mouse click: (88,6) [m_radio_check2=>web]
Check boolean: selected (true) [inputRadio1.Wert1=>web]
Select: 0 [optionNot_selected=>web]
Check boolean: selected (true) [inputRadio1.Wert1=>web]

Figure 10.14: The "Selected test" Node

The node "Selected test" contains one mouse click, two checks and a selection in the middle. The click selects the "Radio buttons & check boxes" page. Afterwards, you see a "Check boolean: selected" node that queries the selected state of the first radio button to determine that it is selected. The "Selection" sets the combo box at the top of the page to the "not selected" option. Finally, another "Check boolean: selected" is performed just as before. You'll see that this last check leads to an error.

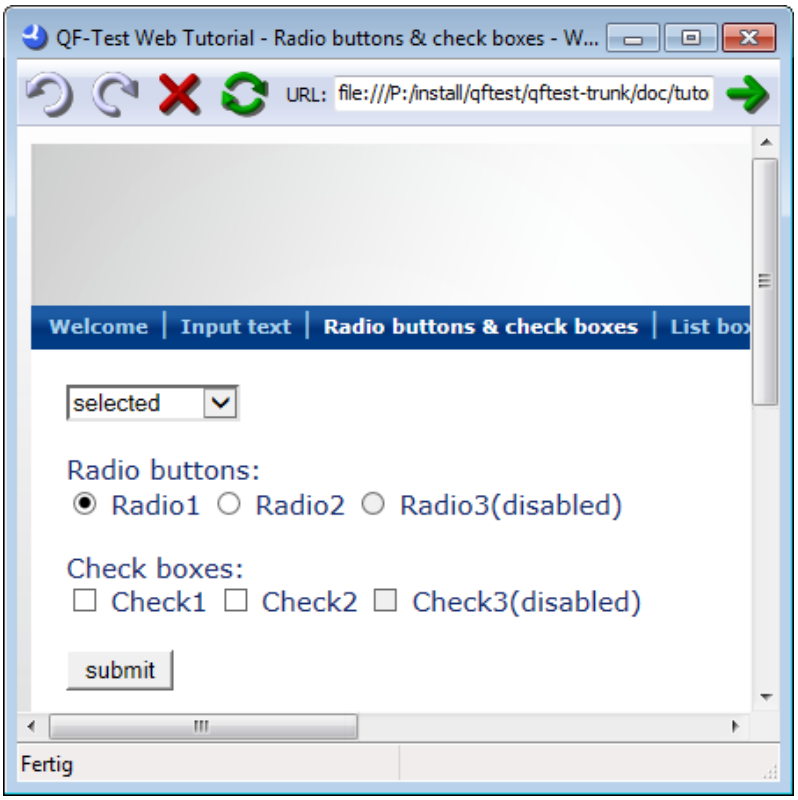

Figure 10.15: Radio Buttons in the Web Demo

Try running the ”Selected test” sequence now so that you can see the actual execution of the description we just provided. Don’t forget to return to the ”Welcome” page first.

10.7 Stopping the Browser

The function of the ”Cleanup” sequence is important to understand. Inside a test-set a ”Cleanup” node will be executed after every test-case located in the same level of the tree-structure, just as the ”Setup” sequence will be executed before each test. The ”Cleanup” node serves the purpose of leaving the SUT in a specific or ’clean’ state so that further tests can be executed deterministically. This step is equivalent to the ”tear-down” method within unit testing.

Expand the ”Cleanup: Stop browser” node now to take a look at its child-nodes. For this cleanup we choose the final way and simply terminate the browser.

Figure 10.16: The Cleanup Sequence ”Stop Browser”

The ”Stop client” node is followed by a ”Wait for process to terminate” node to ensure the next test will wait until the browser is fully terminated.

10.8 The Complete Test

In the previous sections we've taken you step-by-step through many elements of our example test-suite. At this point, you're now ready to execute the entire test in one fell swoop.

First, close the browser in case it is still running. Then select the ”Test-set: Web demo” node and execute it with the replay button. The entire set of tests will last about one minute, due to the delays we built into certain nodes of the test-suite so that you can better follow what's going on. If you look in the details of the ”Test-set: Web demo” node, you'll see a variable called ”delay” defined in the parameter default values section under ”Defaults”. You can change this value if you wish to shorten or lengthen the total run time.

When the tests complete, you should end up with two failures, both of which we discussed in the previous sections. Open up the run-log again to take a closer look:

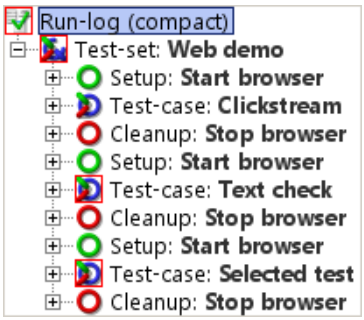

Figure 10.17: Run-Log for the Completed Test-set ”Web demo”

Here you'll see in a visible fashion the process we attempted to explain in the last section concerning the behavior of the test-set node in which execution of the ”Setup” and ”Cleanup” nodes take place before and after each test. Thus a clearly defined ”clean” state is ensured.

10.9 Report Generation

In the world of quality assurance the documentation and archiving of test results are of vital importance. In order to achieve this, QF-Test offers an automated report-generation feature. Since you've just finished a complete test run, we're at a good point to show you the capabilities of this feature.

Make sure the run-log for your test-run is open. Now use the File→Create HTML/XML report... option from the run-log's menu to bring up the dialog window used to specify the nature of your desired report.

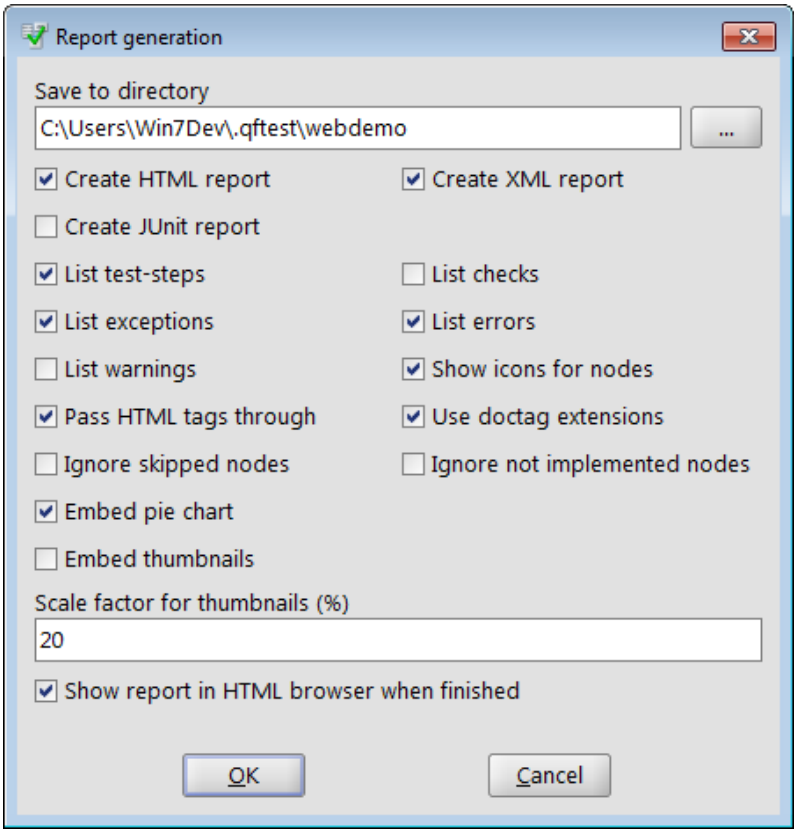

Figure 10.18: Report Generation Properties

In the first field, you can specify the file-name of the report. Following this, you can decide what type of report you want. QF-Test offers two kinds of reports, HTML or XML format. An XML report is useful if you've written your own XSLT stylesheets to shape the report into the style you'd prefer.

There are further options to control handling of HTML and doctags used inside comment fields as well as viewing the report in your browser after creation. You can just

leave those fields unchanged. Further details can be found in chapter Reports and test documentation of the user manual.

Let's try generating a simple HTML report from the results of the last test-run. After possibly adapting the fields we indicated above and confirming the generation with the "OK" button the report will then be generated and generally look as follows in your browser:

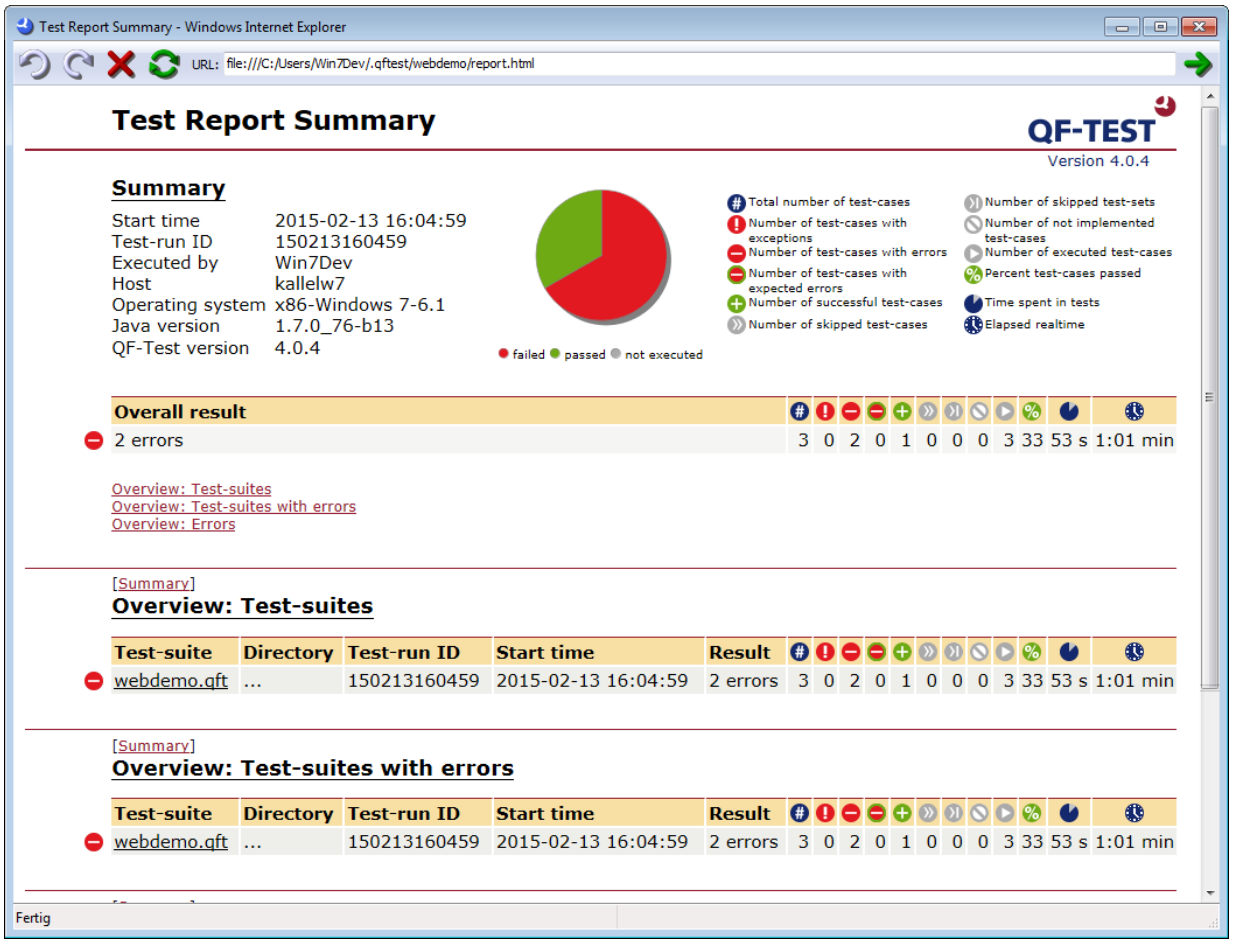
Test Report Summary - Windows Internet Explorer
URL: file:///C:/Users/Win7Dev/.qftest/webdemo/report.html
Test Report Summary
QF-TEST
Version 4.0.4
Summary
Start time 2015-02-13 16:04:59
Test-run ID 150213160459
Executed by Win7Dev
Host kallelw7
Operating system x86-Windows 7-6.1
Java version 1.7.0_76-b13
QF-Test version 4.0.4
failed passed not executed
Total number of test-cases
Number of test-cases with exceptions
Number of test-cases with errors
Number of test-cases with expected errors
Number of successful test-cases
Number of skipped test-cases
Number of skipped test-sets
Number of not implemented test-cases
Number of executed test-cases
Percent test-cases passed
Time spent in tests
Elapsed realtime
Overall result
2 errors 3 0 2 0 1 0 0 0 3 33 53 s 1:01 min
Overview: Test-suites
Overview: Test-suites with errors
Overview: Errors
[Summary]
Overview: Test-suites
Test-suite Directory Test-run ID Start time Result
webdemo.qft ... 150213160459 2015-02-13 16:04:59 2 errors 3 0 2 0 1 0 0 0 3 33 53 s 1:01 min
[Summary]
Overview: Test-suites with errors
Test-suite Directory Test-run ID Start time Result
webdemo.qft ... 150213160459 2015-02-13 16:04:59 2 errors 3 0 2 0 1 0 0 0 3 33 53 s 1:01 min
Fertig

Figure 10.19: An HTML Report

The report begins with a summary containing informational data from your system on the top left side, a legend describing the meaning of icons used in the report on the top right side and the overall test result below. In our case as result we see the two well-known errors again.

Following the summary, an error overview lists the errors with its location where they occurred (the test-case) and the message describing the error details. Part three is a test-set overview just containing the "Web demo" test-set, as our test-suite only contains

that one. Finally all test-cases contained in the ”Web demo” test-set are listed with details like description, result and execution time.

The report generation feature proves very useful for creating an overview over the test-run and a document for presentation and archiving purposes.

Chapter 11

Creating a Test-suite [45-60 min]

11.1 Introduction

This chapter will take you through the steps necessary for creating your first test-suite:

- Starting the SUT from within QF-Test.
- Recording some actions and organizing your test-suite.
- Creation of tests which checks real functionality within the SUT.

11.2 Starting the Application

Open a new (empty) test-suite from QF-Test with the File→New Suite menu option. Please be aware the ”old” test-suite remains open as a different window.

Make sure that the detailed view is turned on, it can be toggled with the View→Show Details menu option. You now see the divided window with a tree on the left-hand side and a pane on the right-hand side representing the details of a selected node. Using the tree you can navigate through the test-suite’s nodes; when you select a node, you’ll see its properties appear within the details pane on the right.

At the beginning the application to be tested needs to be started from QF-Test. In this case it will be the browser showing the ”Web demo” page which is already known from the previous chapter. The QF-Test Quickstart Wizard will help us to create an appropriate setup sequence.

Please open the Quickstart Wizard via the Extras menu. He should welcome you as shown in the figure below. After saying a short hello please press the ”Next” button.

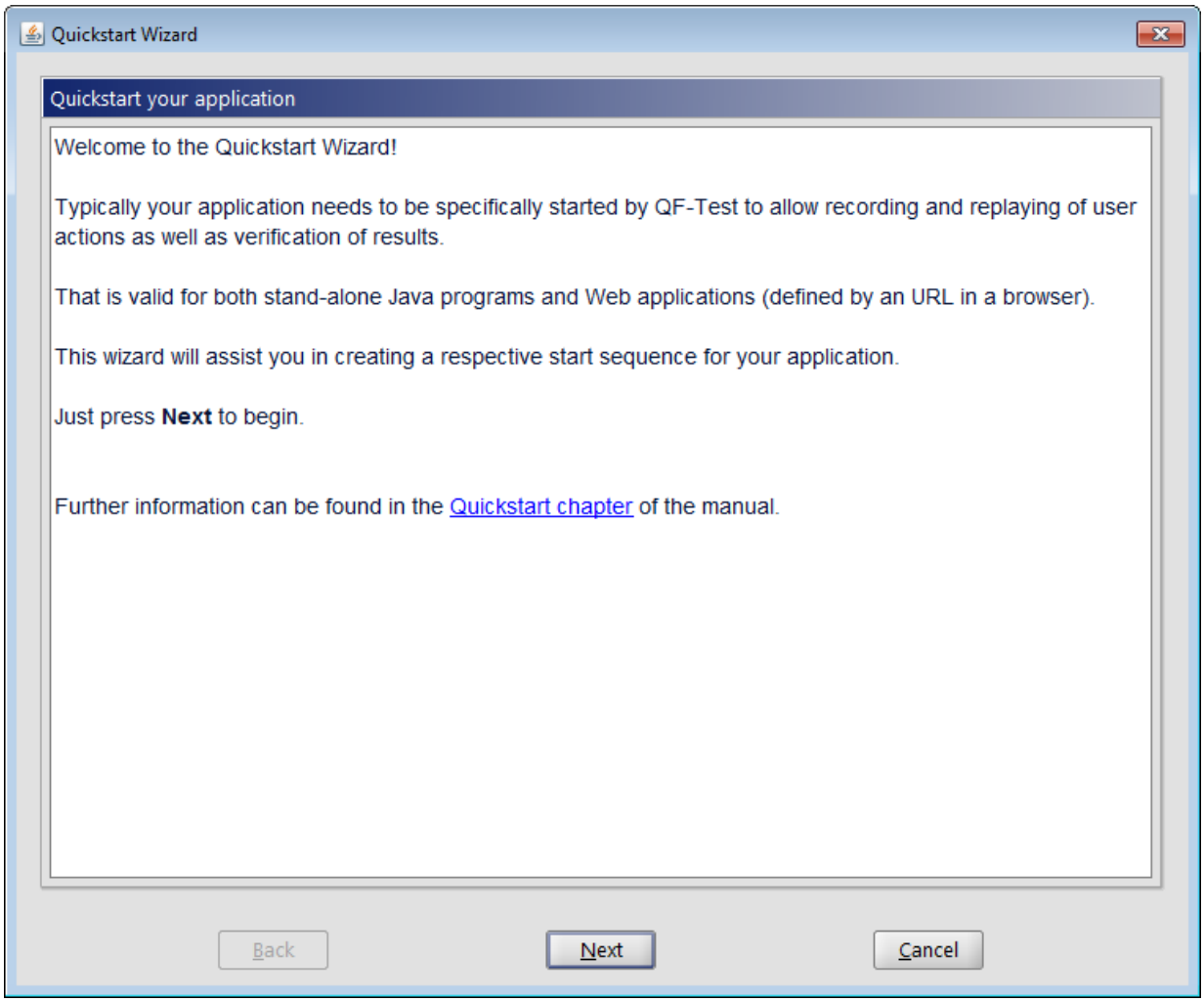
Quickstart Wizard
Quickstart your application
Welcome to the Quickstart Wizard!
Typically your application needs to be specifically started by QF-Test to allow recording and replaying of user actions as well as verification of results.
That is valid for both stand-alone Java programs and Web applications (defined by an URL in a browser).
This wizard will assist you in creating a respective start sequence for your application.
Just press Next to begin.
Further information can be found in the Quickstart chapter of the manual.
Back
Next
Cancel

Figure 11.1: The Quickstart Wizard

Next you can choose the type of application to be tested. Please select ”A web page in a browser” and continue.

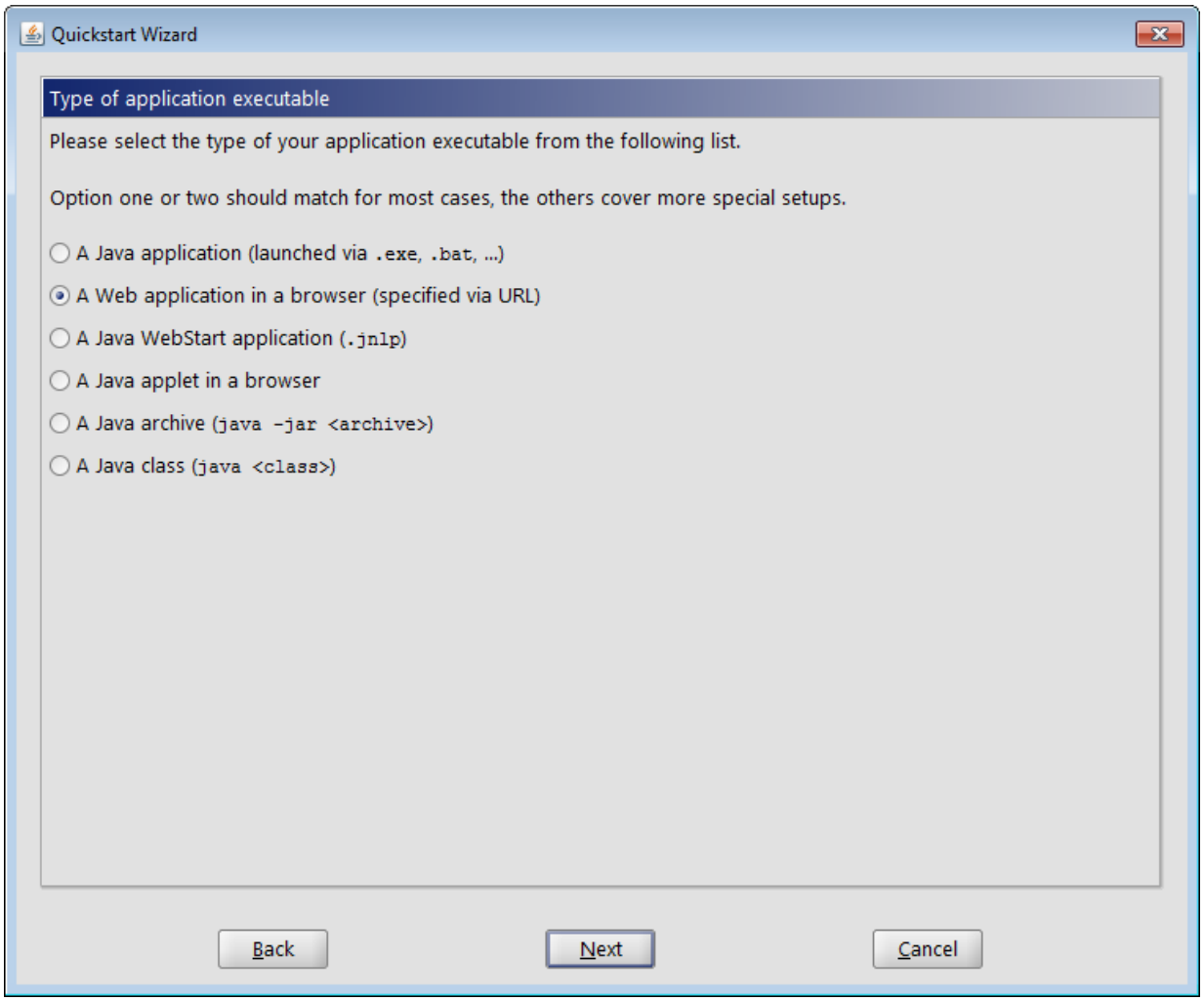
Quickstart Wizard
Type of application executable
Please select the type of your application executable from the following list.
Option one or two should match for most cases, the others cover more special setups.
A Java application (launched via .exe, .bat, ...)
A Web application in a browser (specified via URL)
A Java WebStart application (.jnlp)
A Java applet in a browser
A Java archive (java -jar <archive>)
A Java class (java <class>)
Back
Next
Cancel

Figure 11.2: Select type of SUT

Now you are asked about the URL of the web page to be tested which will be the ”Web demo” page. The simplest way to enter it is to open the file browser by clicking the button on the right side of the text field and navigate to the qftest-x.y.z version specific directory and from there to `.../doc/tutorial/web/en` where you can select the `index.html` file. Afterwards please proceed to the next wizard step.

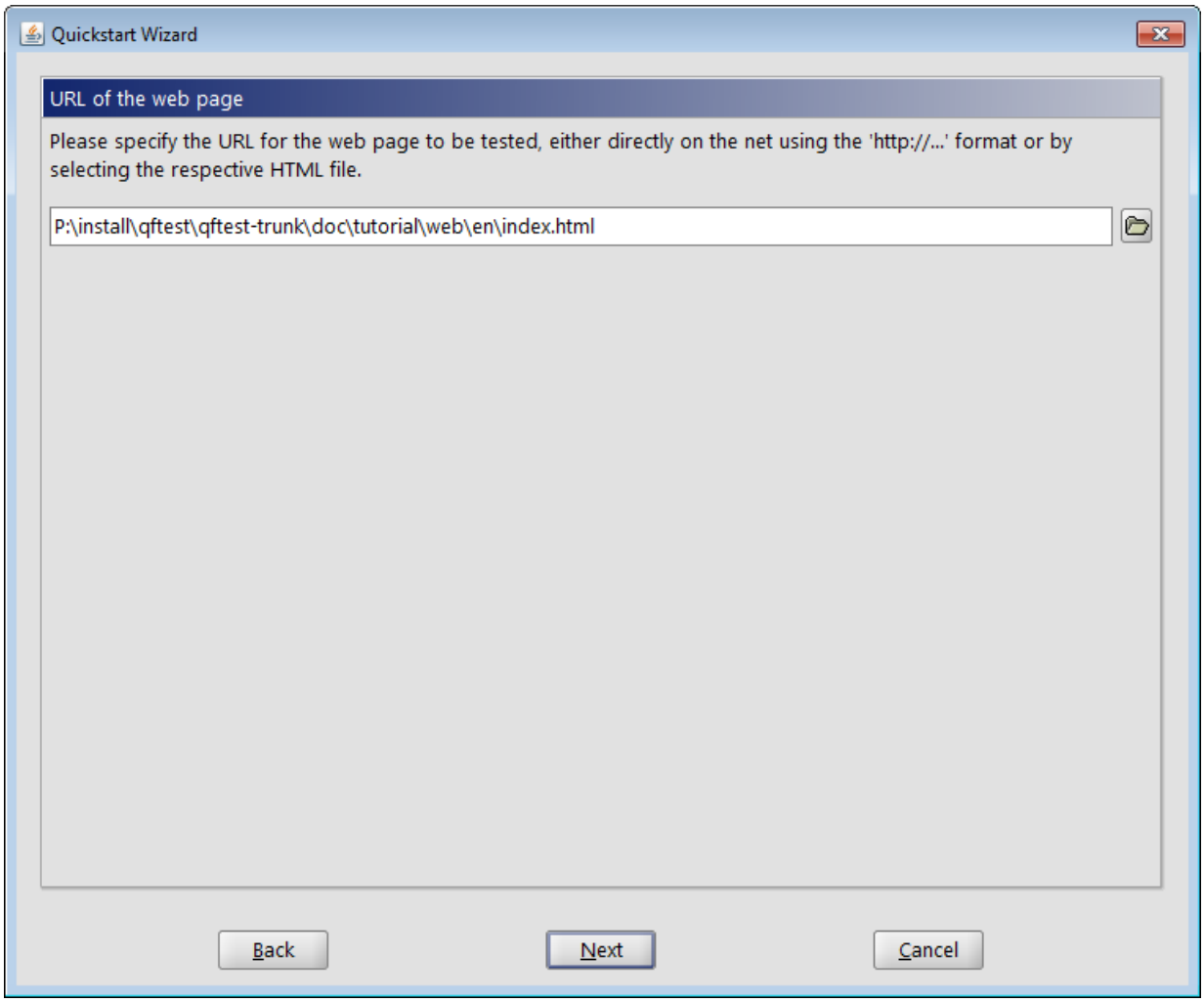
Quickstart Wizard
URL of the web page
Please specify the URL for the web page to be tested, either directly on the net using the 'http://...' format or by selecting the respective HTML file.
P:\install\qftest\qftest-trunk\doc\tutorial\web\en\index.html
Back
Next
Cancel

Figure 11.3: Specification of the URL.

On the next pane you are asked whether your web application to be tested is based on an AJAX toolkit. Our demo does not use one of those toolkits but we can simply keep the recommended auto detection and proceed to the next wizard step.

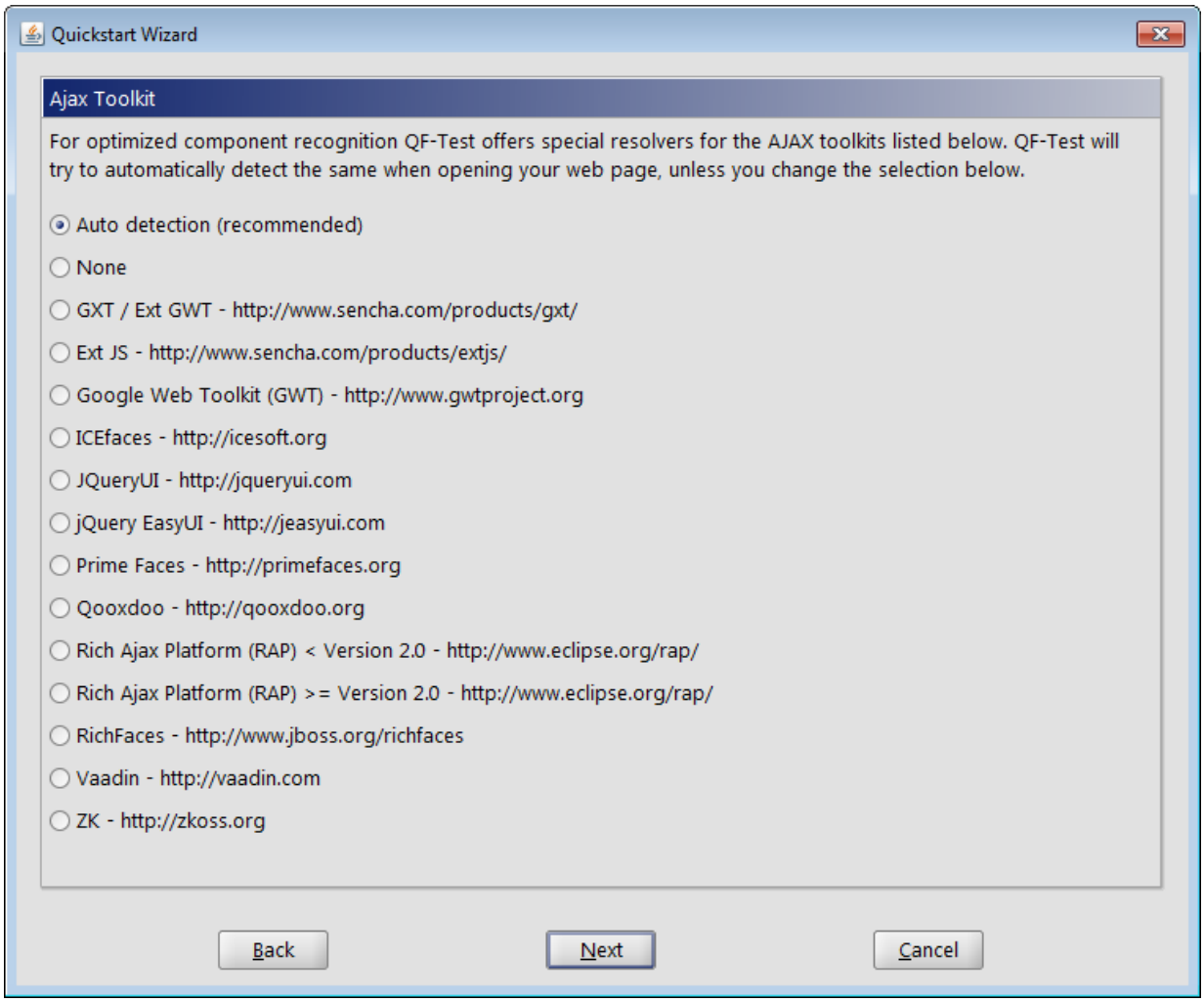
Quickstart Wizard
Ajax Toolkit
For optimized component recognition QF-Test offers special resolvers for the AJAX toolkits listed below. QF-Test will try to automatically detect the same when opening your web page, unless you change the selection below.
Auto detection (recommended)
None
GXT / Ext GWT - http://www.sencha.com/products/gxt/
Ext JS - http://www.sencha.com/products/extjs/
Google Web Toolkit (GWT) - http://www.gwtproject.org
ICEfaces - http://icesoft.org
JQueryUI - http://jqueryui.com
jQuery EasyUI - http://jeasyui.com
Prime Faces - http://primefaces.org
Qooxdoo - http://qooxdoo.org
Rich Ajax Platform (RAP) < Version 2.0 - http://www.eclipse.org/rap/
Rich Ajax Platform (RAP) >= Version 2.0 - http://www.eclipse.org/rap/
RichFaces - http://www.jboss.org/richfaces
Vaadin - http://vaadin.com
ZK - http://zkoss.org
Back
Next
Cancel

Figure 11.4: AJAX toolkit.

It's time now to select the browser to use. On Windows it's most likely the Internet Explorer which is already preselected and we will choose that one for our first own test. On Linux you may chose Firefox.

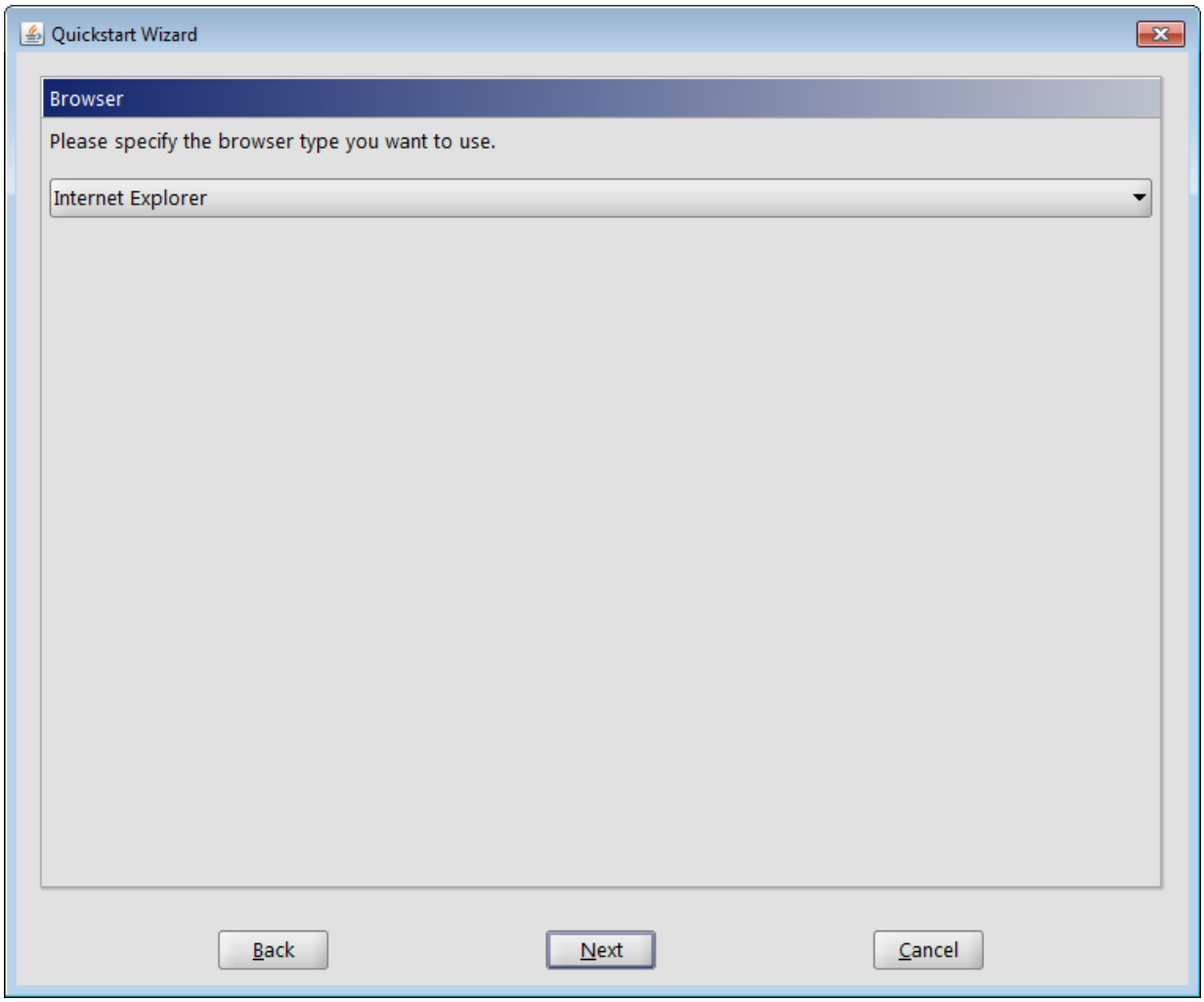

Figure 11.5: Browser selection.

On this pane we have the option to specify a special Internet Explorer compatibilty mode in case the web page to be tested only works properly with a special IE version. As this is not necessary in our case we simply proceed to the next wizard step. (If you had Firefox selected, you will see a different pane where you need to provide the installation directory of the Firefox you want to use.)

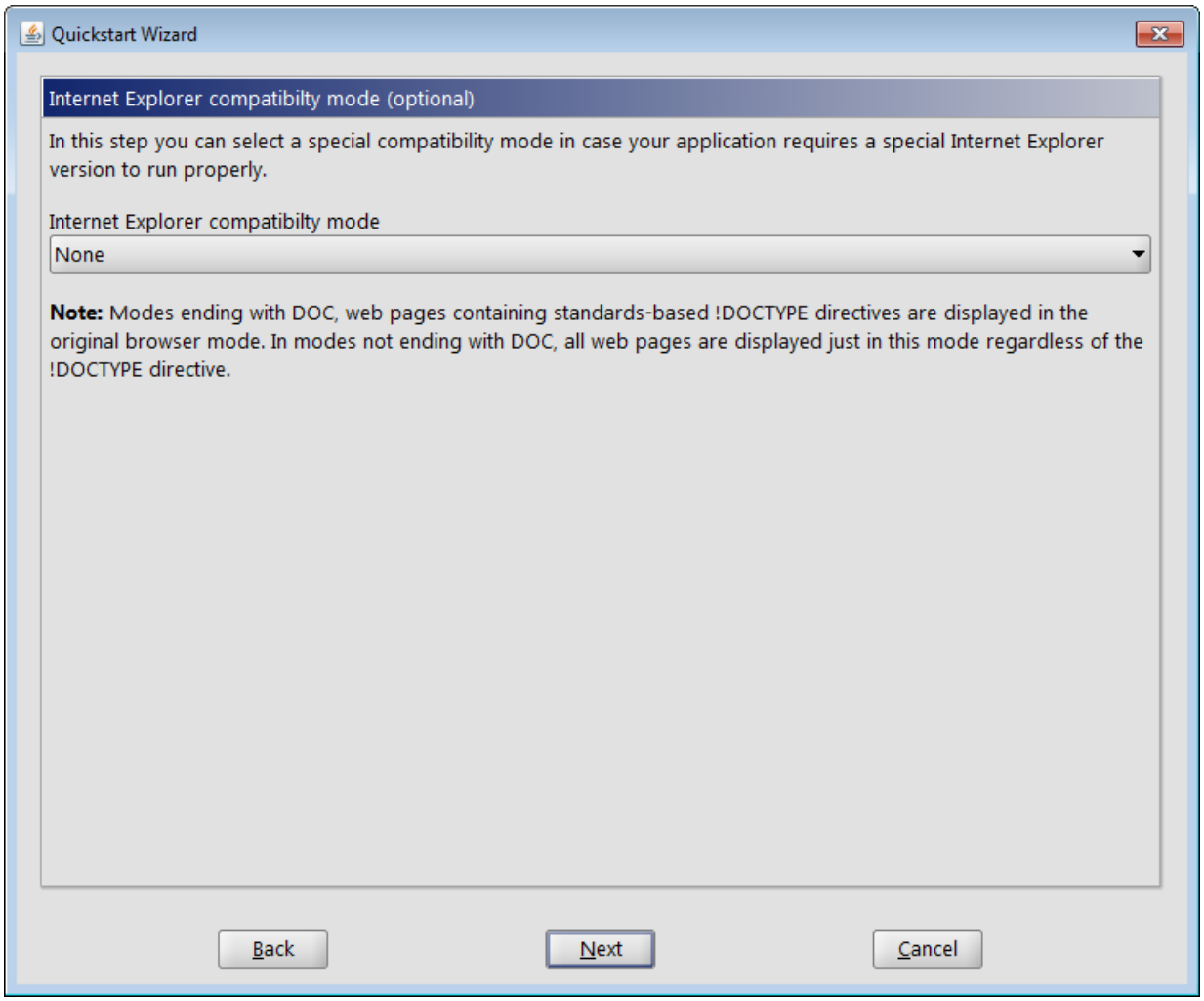
Quickstart Wizard
Internet Explorer compatibilty mode (optional)
In this step you can select a special compatibility mode in case your application requires a special Internet Explorer version to run properly.
Internet Explorer compatibilty mode
None
Note: Modes ending with DOC, web pages containing standards-based !DOCTYPE directives are displayed in the original browser mode. In modes not ending with DOC, all web pages are displayed just in this mode regardless of the !DOCTYPE directive.
Back
Next
Cancel

Figure 11.6: Internet Explorer compatibility mode.

The next step provides the option to start the browser wrapper by use of a dedicated Java version. This may be necessary in case you have multiple Java versions installed on your machine (i.e. 32bit and 64bit) or if Java applets are involved. In most cases you might simply be able to continue with the default value.

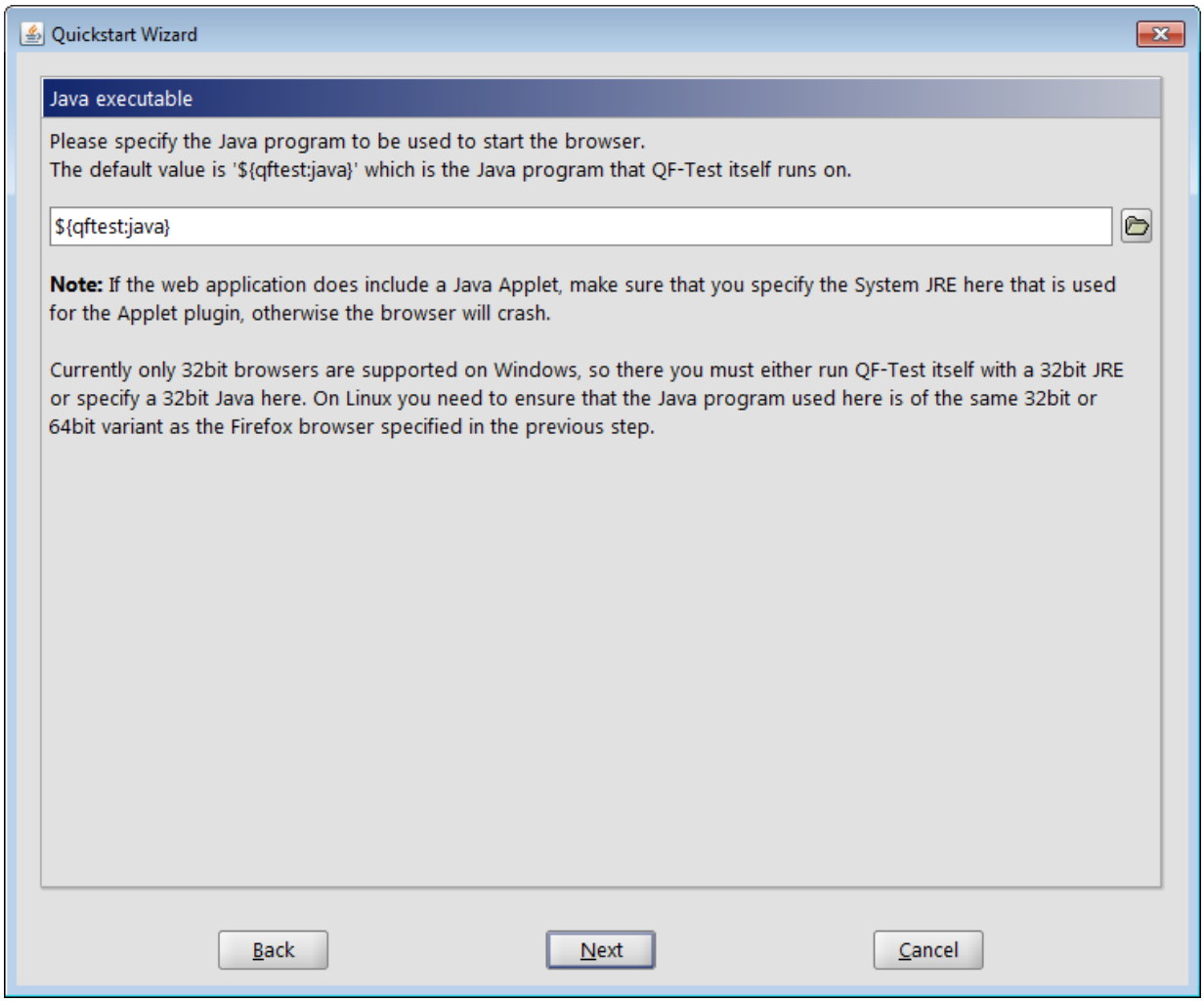
Quickstart Wizard
Java executable
Please specify the Java program to be used to start the browser.
The default value is '${qftest:java}' which is the Java program that QF-Test itself runs on.
${qftest:java}
Note: If the web application does include a Java Applet, make sure that you specify the System JRE here that is used for the Applet plugin, otherwise the browser will crash.
Currently only 32bit browsers are supported on Windows, so there you must either run QF-Test itself with a 32bit JRE or specify a 32bit Java here. On Linux you need to ensure that the Java program used here is of the same 32bit or 64bit variant as the Firefox browser specified in the previous step.
Back
Next
Cancel

Figure 11.7: Java executable for starting the Browser wrapper.

This step gives us the option for some basic browser settings. You can simply keep the proposed settings and continue.

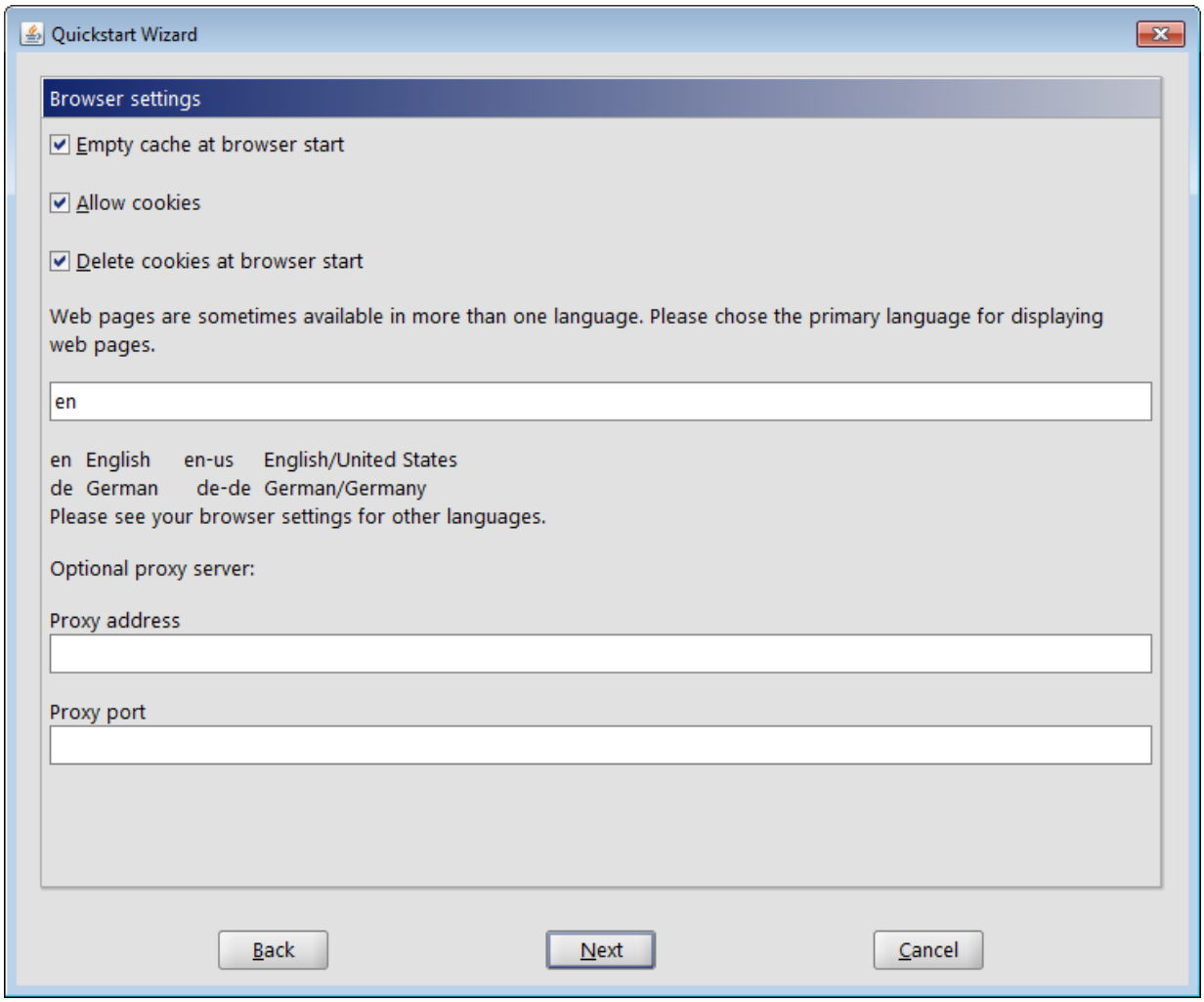

Figure 11.8: Browser settings

As we only want to work with one browser window, you can also continue at the next step without any change.

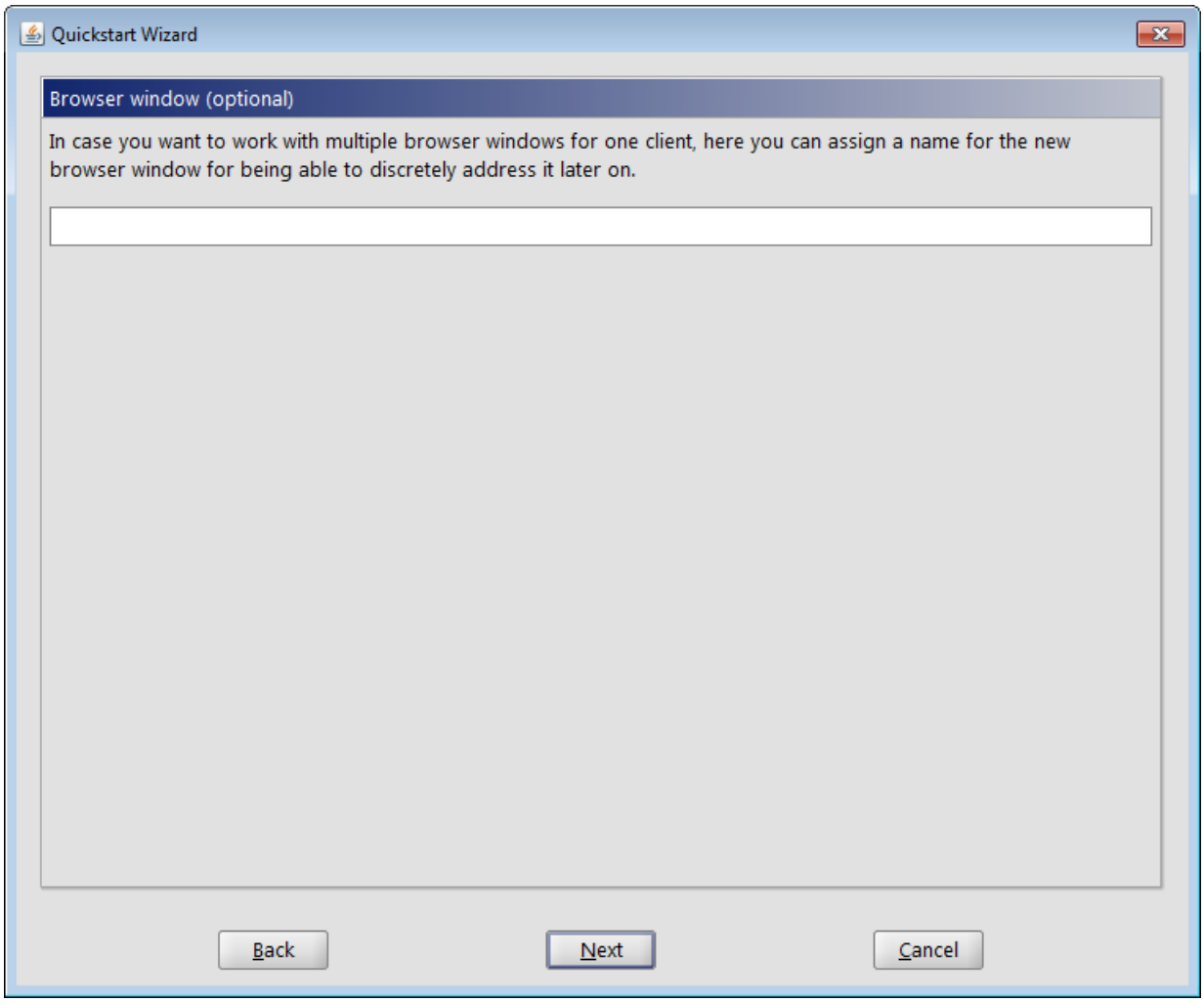
Quickstart Wizard
Browser window (optional)
In case you want to work with multiple browser windows for one client, here you can assign a name for the new browser window for being able to discretely address it later on.
Back
Next
Cancel

Figure 11.9: Browser window name

It's nearly done. We have just to assign a name to our client as a reference. Let's simple call it "web". Please let the check box on the page untouched empty and press "Next".

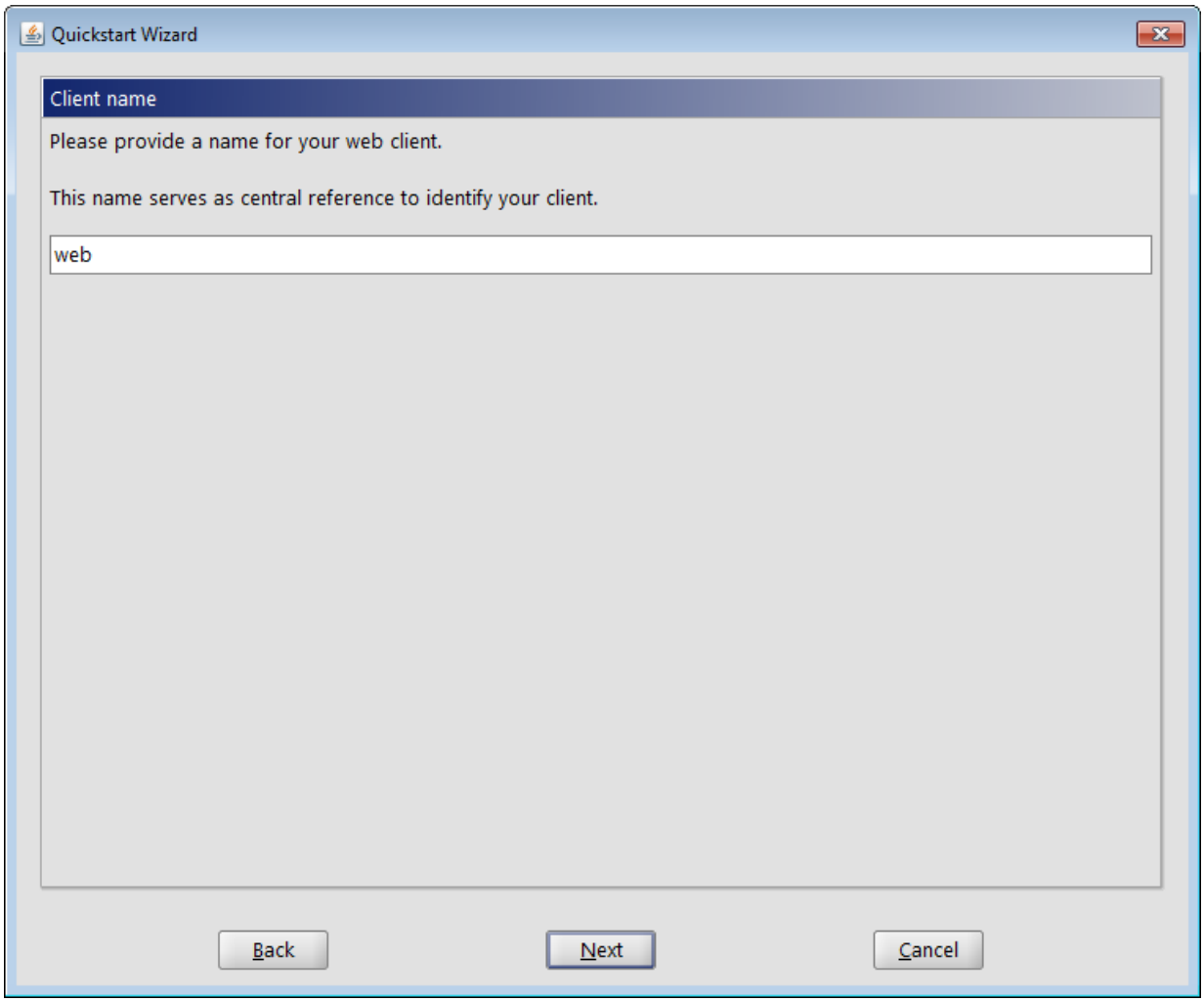

Figure 11.10: Client name

Finally some information is given about what to expect when the wizard finishes its task of creating a setup sequence and where to find help in case of troubles. Please disable the ”Start automatically” option, as we first want to take a look at the generated setup sequence and start it by hand. Now please press ”Finish”.

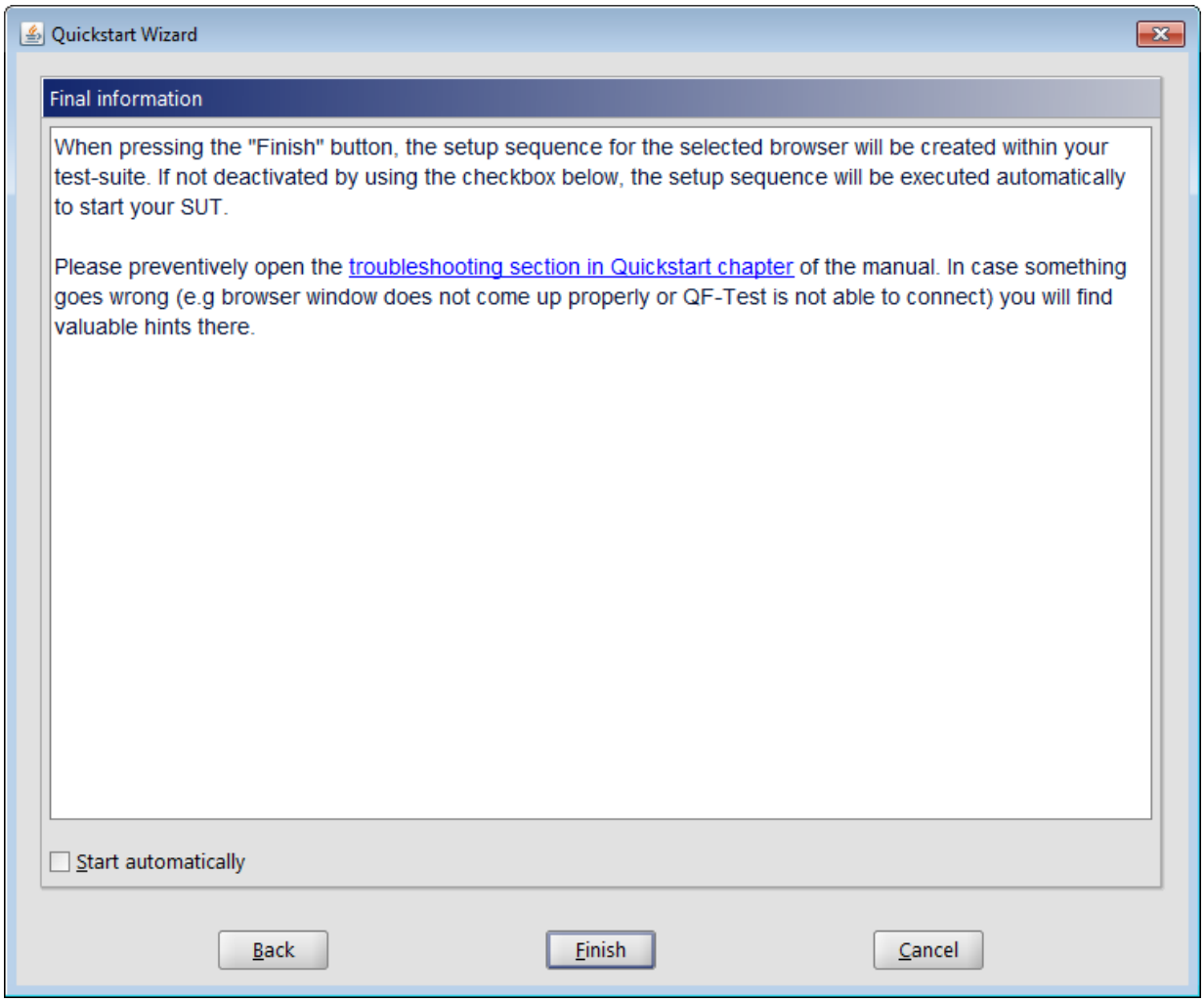
Quickstart Wizard
Final information
When pressing the "Finish" button, the setup sequence for the selected browser will be created within your test-suite. If not deactivated by using the checkbox below, the setup sequence will be executed automatically to start your SUT.
Please preventively open the troubleshooting section in Quickstart chapter of the manual. In case something goes wrong (e.g browser window does not come up properly or QF-Test is not able to connect) you will find valuable hints there.
Start automatically
Back
Finish
Cancel

Figure 11.11: Final Information

The generated setup sequence appearing under the ”Extras” contains four main steps:

- Set global variables
- Start browser without window
- Perform browser settings
- Open browser window with URL

This might look a bit circumstantial in the first place but leaves room for possible required adaptations in a direct way. Expanding the four seqences reviels the details of each step.

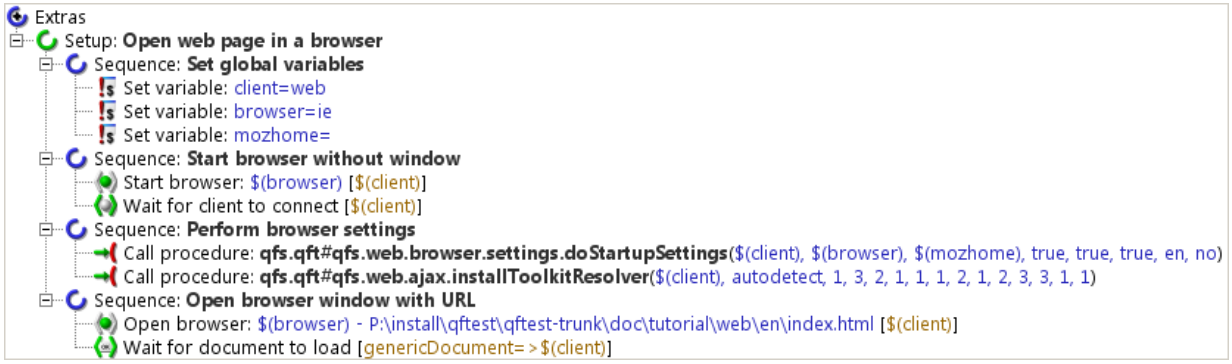
Extras
Setup: Open web page in a browser
Sequence: Set global variables
Set variable: client=web
Set variable: browser=ie
Set variable: mozhome=
Sequence: Start browser without window
Start browser: $(browser) [$(client)]
Wait for client to connect [$(client)]
Sequence: Perform browser settings
Call procedure: qfs.qft#qfs.web.browser.settings.doStartupSettings($(client), $(browser), $(mozhome), true, true, true, en, no)
Call procedure: qfs.qft#qfs.web.ajax.installToolkitResolver($(client), autodetect, 1, 3, 2, 1, 1, 1, 2, 1, 2, 3, 3, 1, 1)
Sequence: Open browser window with URL
Open browser: $(browser) - P:\install\qftest\qftest-trunk\doc\tutorial\web\en\index.html [$(client)]
Wait for document to load [genericDocument=>$(client)]

Figure 11.12: Generated Setup Sequence

- The first step contains three ”Set variable” nodes to define global variables: ”client”, the identifier to address the client, the browser type ”browser” and ”mozhome”, the installation directory necessary for Mozilla based browsers but not for Internet Explorer and therefore empty in our example.
- In step two the browser process is started, but without opening a browser window. There is also a wait step to ensure the client is running until the test goes on.
- Next step is to perform some browser settings by calling the ”doStartupSettings” procedure from the `qfs.qft` standard library with respective parameters. Also the AJAX autodetect resolver gets installed.
- And in the final step the browser window is opened with the given URL followed by another wait step to ensure the document is fully loaded before continuing.

You can have a look at the details of some of the node if you like. Basically you will find the values you provided during the Wizard steps assigned to the respective node attributes.

Now we want to see some action. Please ensure the ”Setup” node is selected. Then click the replay button or simply hit ”Enter”.

After a few second the browser window is expected to appear showing the web demo page similar to figure 10.5(126) from the previous chapter:

11.3 Adding a Clickstream Test

You’re now ready to add a simple clickstream test.

- We want to start the recording from the ”Welcome” page. Please ensure it is selected.

- In QF-Test now press the ”Start recording” button and switch to the browser window. From now on every mouse and keyboard action performed within the SUT window will be recorded.
- Now please change in the browser to the ”Radio buttons & check boxes” page (see figure 10.15(134) in previous chapter) and perform some actions there and conclude by pressing the ”submit” button.
- Finally switch back to your QF-Test test-suite window. Press ”Stop recording” . You’ll find the recorded sequence placed under ”Extras” node on the left side of the main window, as seen below:

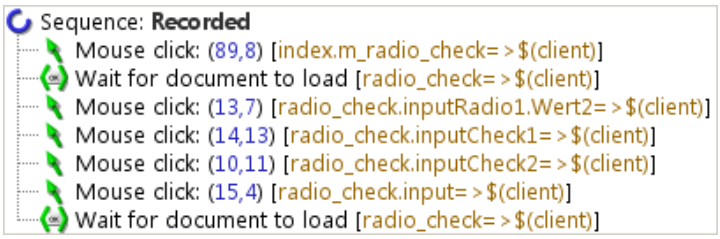
Sequence: Recorded
Mouse click: (89,8) [index.m_radio_check=>$(client)]
Wait for document to load [radio_check=>$(client)]
Mouse click: (13,7) [radio_check.inputRadio1.Wert2=>$(client)]
Mouse click: (14,13) [radio_check.inputCheck1=>$(client)]
Mouse click: (10,11) [radio_check.inputCheck2=>$(client)]
Mouse click: (15,4) [radio_check.input=>$(client)]
Wait for document to load [radio_check=>$(client)]

Figure 11.13: The Recorded Clickstream

The name of the recorded sequence will be entered in a standard fashion by QF-Test as the time and date of the recording. You can change this name as you see fit by simply clicking on the node and changing its properties in the details view on the right.

Inside the sequence, beside ”Mouse click” nodes there are also two ”Wait for document to load” nodes. QF-Test creates such automatically when a new document gets loaded. In our case that’s on the initial change to the ”Radio buttons & check boxes” page and after pressing the ”submit” button, which also cause a reload.

We want to replay the newly created sequence right away for verification. In advance we may not forget to change to the ”Welcome” page as this was the starting point of the recording. Then we select the sequence node and execute it using the button. You should now see your exact sequence of mouse and keyboard events replayed within the browser window.

Before creating a more complex test we will first structure the recorded sequences into a test-suite.

11.4 Building a Test-suite

So far we have worked in the ”Extras”, which is a kind of sandbox for experimenting. Now we want to create a real test-suite.

At first please select the ”Test-set” node an change its name to e.g. ”Webdemo”. Next expand the node and change the name to the enclosed ”Test-case” node e.g. to ”Clickstream”.

Then we need to move the setup sequence from the ”Extras” into the test-set, at the first position to be more specific, that means before the ”Clickstream” test-case. Moving the ”Setup” node can be accomplished with mouse (Drag&Drop), context-menu (use a click of the right mouse-button) or with the Control-X (cut) and Control-V (paste) keyboard commands.

Note When using Drag&Drop a target node can be expanded by having the mouse cursor hovered above the ”+” left of the target node.

Now it’s time to move your recorded Clickstream sequence from the ”Extras” node up into the ”Clickstream” test-case node. If the sequence still has the time stamp as name, you might want to turn it into something more meaningful.

Lastly you can create a ”Cleanup” sequence to stop the application. This sequence contains two nodes: One to stop the client and one to ensure that it actually terminated. From the figure below, you should be able to see what nodes are necessary to carry out this little exercise (you might want to look under ”Dependencies” and ”Process nodes”).

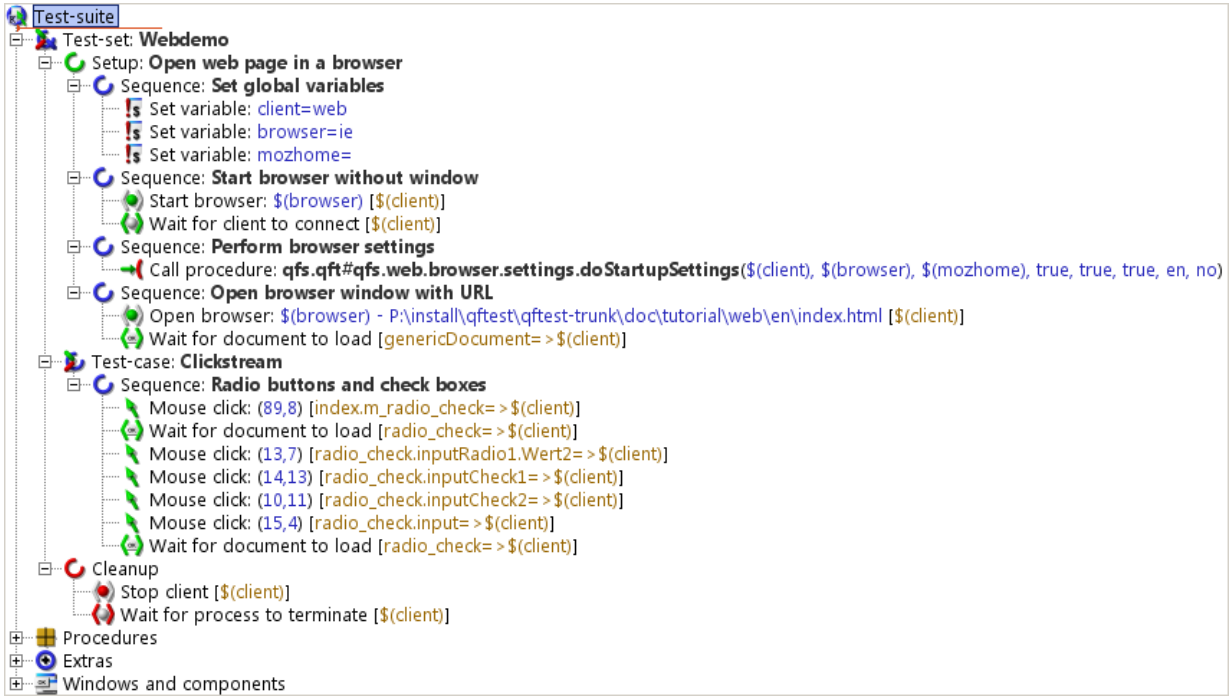
Test-suite
Test-set: Webdemo
Setup: Open web page in a browser
Sequence: Set global variables
Set variable: client=web
Set variable: browser=ie
Set variable: mozhome=
Sequence: Start browser without window
Start browser: $(browser) [$(client)]
Wait for client to connect [$(client)]
Sequence: Perform browser settings
Call procedure: qfs.qft#qfs.web.browser.settings.doStartupSettings($(client), $(browser), $(mozhome), true, true, true, en, no)
Sequence: Open browser window with URL
Open browser: $(browser) - P:\install\qftest\qftest-trunk\doc\tutorial\web\en\index.html [$(client)]
Wait for document to load [genericDocument=>$(client)]
Test-case: Clickstream
Sequence: Radio buttons and check boxes
Mouse click: (89,8) [index.m_radio_check=>$(client)]
Wait for document to load [radio_check=>$(client)]
Mouse click: (13,7) [radio_check.inputRadio1.Wert2=>$(client)]
Mouse click: (14,13) [radio_check.inputCheck1=>$(client)]
Mouse click: (10,11) [radio_check.inputCheck2=>$(client)]
Mouse click: (15,4) [radio_check.input=>$(client)]
Wait for document to load [radio_check=>$(client)]
Cleanup
Stop client [$(client)]
Wait for process to terminate [$(client)]
Procedures
Extras
Windows and components

Figure 11.14: The Organization of your Test-suite

Now you’ve completed the most important steps of structuring a test-suite. In the following section you’ll extend your test-suite by introducing a check of a specific text-field.

11.5 Adding a Text Check

Let's create a little more complex test in which we will fill a form and check the result string.

- In case the browser isn't running please execute the setup sequence. Otherwise switch back to the "Welcome" page to have a proper initial state. In QF-Test now press the "Start recording" button and switch back to the browser window.
- Select the "Input text" page via the menu bar. We know the page from chapter section 10.5[(128)].
- Then please fill in the form with some data and press the "submit" button. A string is printed in the text field at the bottom of the page, containing a summary of the given data.
- We want to record a check for the result string. For this please click in QF-Test the "Record a check" button and switch to the browser window. Now when you move your mouse over the components they get highlighted by a frame. To record the check, move to the respective text area at the bottom and right-click. You'll then see a popup menu appear which gives you the choice of several checks, starting with "Text", as seen below:

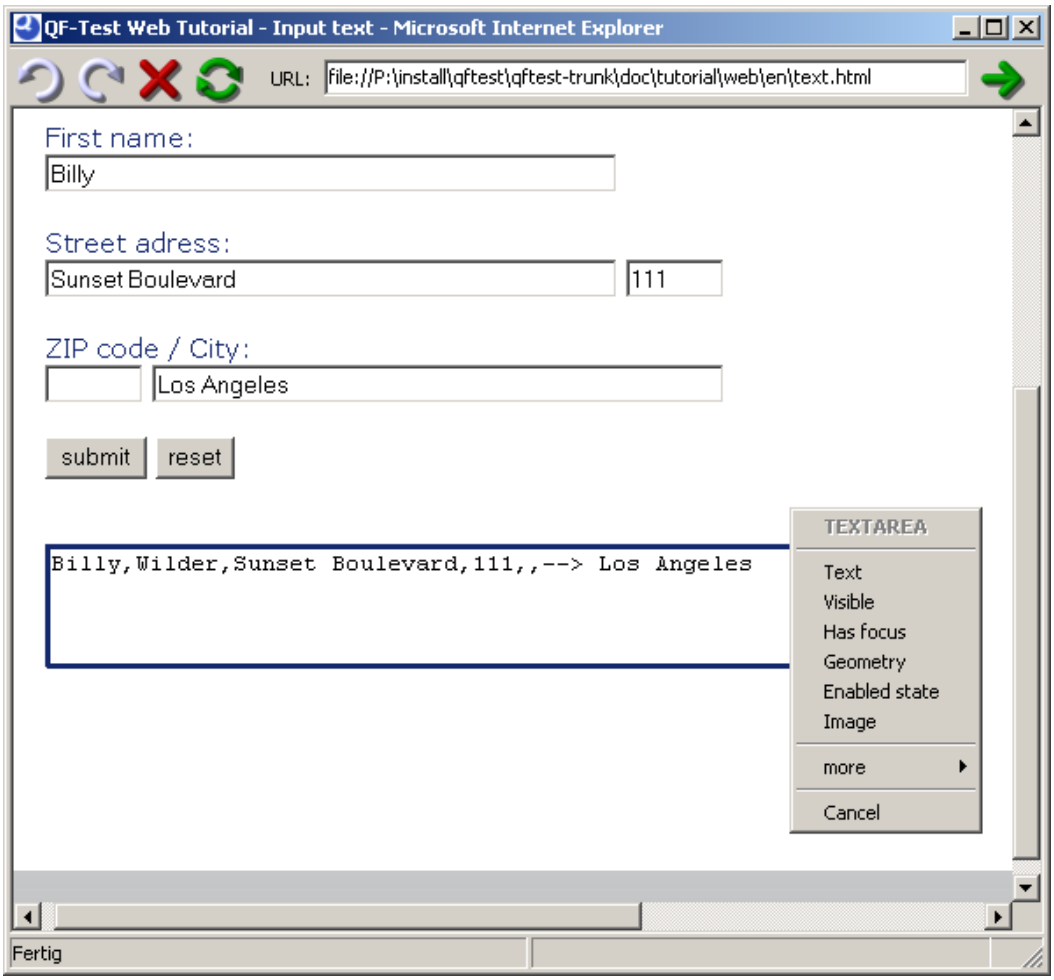

Figure 11.15: Recording a Text Check

- Choose ”Text” from the popup menu, return to your test-suite window and stop recording using the ■ button.

Note

Instead of returning to the test-suite to activate the check mode by pressing the ✓ button, you can also utilize F12 while remaining in the SUT. This hotkey activates/deactivates the record check mode.

You should now have the know-how to organize the recorded sequences into a test-case. You can compare your results with the way we organized the suite and renamed the sequences. Additionally we transformed sequences into ”Test-step” nodes which specify them more clearly and make them also visible in the test report.

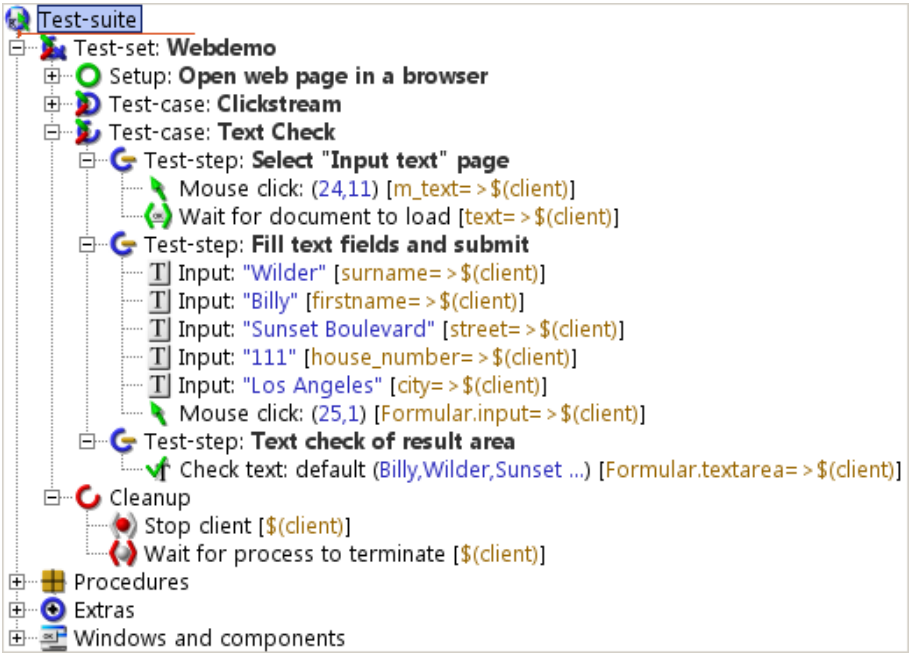

Figure 11.16: The Text-Check Test Organized into the Tree Structure

Note

Transformation of nodes can be easily done via menu Operations | Transform node into or also a nodes context menu.

At this point, you can give your new test a shakedown run. Stop the SUT client now if it is running, click on the root test-node, and start execution by hitting the replay button. Does anything work as expected?

The result of the test-run is stored in the run-log. To view it just make use of the error dialogs "Open run-log" button or use the Run→1. ... menu option or take the short-cut Control-L.

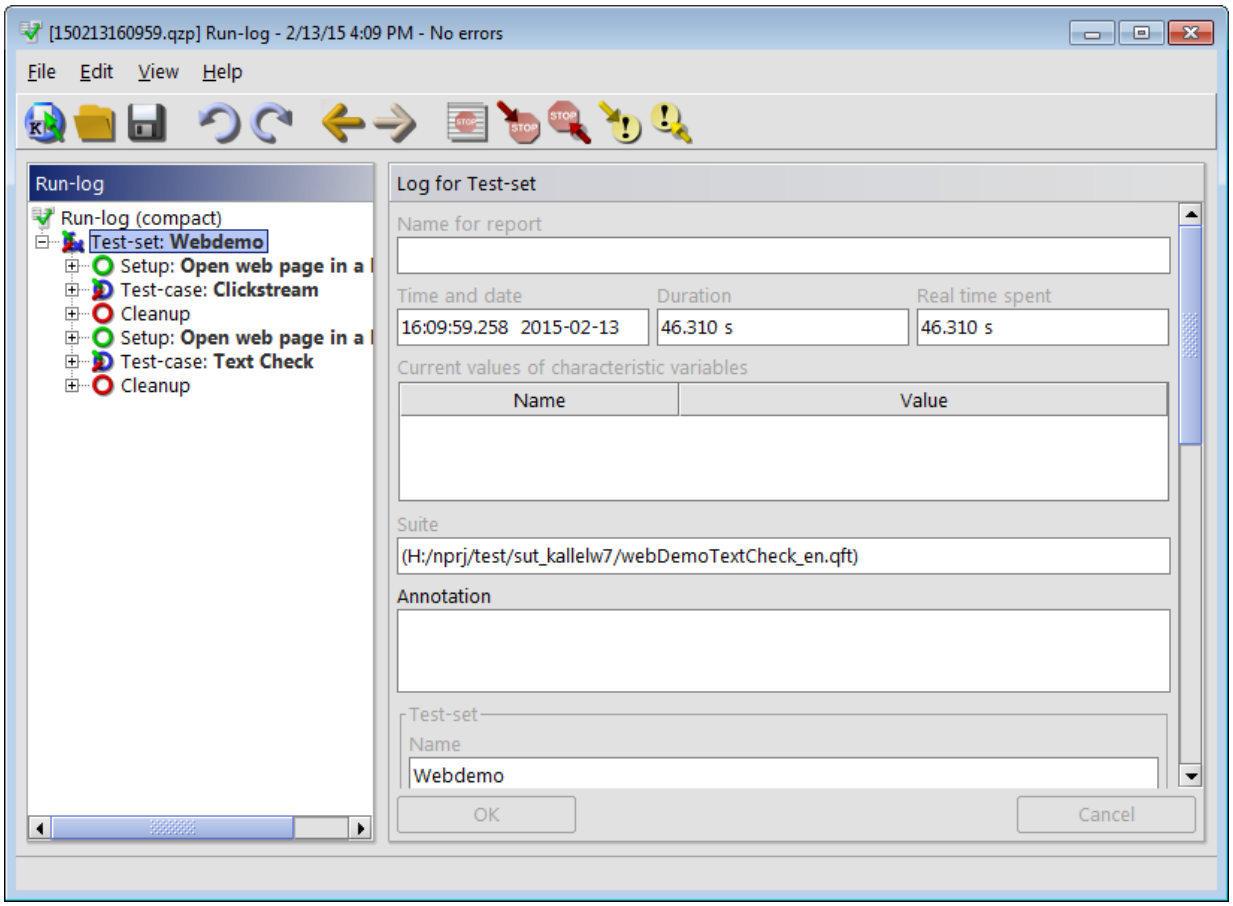

Figure 11.17: The Run-Log of Your Test

Within the run-log you see that the ”Setup”/”Cleanup” pair is executed before and after each test-case.

Note

Starting and stopping the SUT before and after each test-case is a safe way for a proper state but a time consuming one. Typically you will try to start the SUT only once and use others methods for resetting it to a defined status.

In this test there were no errors or exceptions in the test-run. But you can see yellow frames around some nodes, which indicate warnings. In case of the webdemo they are caused by components which have no name assigned. We will not dicuss those warnings here any further, but you will find detailed information on the importance of component names in the user manual.

Upon closing the run-log window you will be prompted if you want to save the run-log (since it will only be available until QF-Test ends).

Let’s modify the test now so that we cause an error in the text-check. Click on your ”Check Text” node so that you see its details displayed:

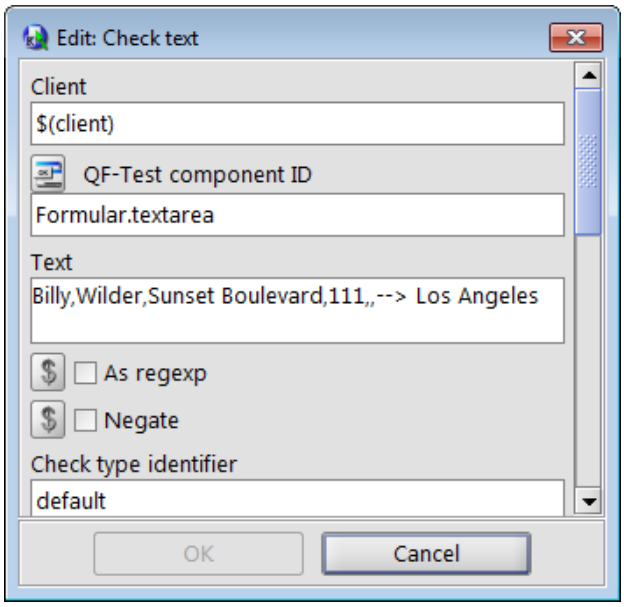

Figure 11.18: Properties of the ”Check Text” Node

In the ”Text” field you see the value that QF-Test expects. Change this text to some other arbitrary value and re-run the test.

Now when you replay the test, a dialog box appears at the end indicating that there was an error.”

If you open the run-log, you’ll see that red-fields now enclose certain nodes, indicating that some error occurred inside the child-nodes. You can use the run-log menu option Edit→Find next error to jump directly to the node which caused the error or use the Control-N keyboard shortcut.

To correct your check node again you can use a helpful feature in QF-Test. In the run-log simply click right on the node indicating the mismatch and select ”Update check nod with current data” from the context menu.

We did it. Your first test-suite is completed. Finally you might want to save it.

Chapter 12

Further reading [5 min]

This part on web testing only covers the first initial steps. As concepts and way of working are basically the same, please also make use of the parallel comprehensive part on Java testing.

In addition, an enhanced suite `webdemo2.qft` for the web demo used in this tutorial part can be found in the directory `qftest-4.0.8/demo/web/`. Advanced techniques like dependencies, data-driver and even resolvers are used. Also you can see how multiple browsers on different operating systems can be used for testing.

Part III

Advanced features of QF-Test

Chapter 13

Introduction [5 min]

This part of the tutorial covers advanced QF-Test features. The following chapters use the JCarConfigurator as example which is a Java/Swing application, but the concepts of the topics is the same for each technology.

We also provide dedicated test-suite files for each chapter, so that you get the chance to take a look at the single topics separately. These files are located in `qftest-4.0.8/doc/tutorial/ advanced-demos/en`.

Chapter 14

The Demo Application [5 min]

The SUT for the advanced tutorial is an application called JCarConfigurator. It allows the user to calculate prices for cars. Here is a list of features the application should cover:

- Select a model of a car and calculate the base price.
- Assign some specific accessories to the selected car.
- Collect several accessories as a special model.
- Grant discount on the final price.
- Create a new car model.
- Create a new accessory.
- Create a new special model.

You can start the JCarConfigurator either via `qftest-4.0.8/demo/carconfig/run_car.bat` on Windows or `qftest-4.0.8/demo/carconfig/run_car.sh` on Unix systems.

Note You have to pass a parameter `en` to the script call to run the SUT in English. Here is a screen shot of the main-window:

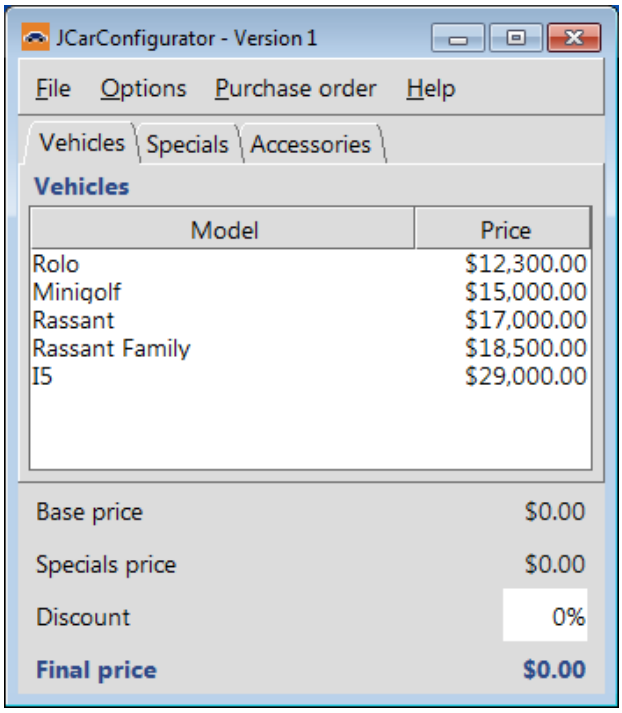

Figure 14.1: The main window of the JCarConfigurator

Note The JCarConfigurator is small and implemented in Java/Swing, but the concepts of testing this application are the same than testing a huge application in any technology QF-Test supports.

Chapter 15

Data driven Testing: Running one Test-case with different test data [30-45 min]

This chapter explains how data driven testing can be achieved using QF-Test.

You can find the implementation of the following examples in the demo test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/datadrivenTesting.qft`.

15.1 Situation

The users of the JCarConfigurator can grant several discount levels to their customers. Thus the test designers have figured out three discount levels which have to be tested.

Those discount levels are 0%, 10% and 15%.

As the workflow of granting discounts is the same for each discount level, we can use the same test-case for testing them. The only difference between the test-cases is the input of the actual discount level and the price to check. A major advantage of using the same test-case is that this avoids side effects caused by several implementations of a Test-case, which could have different bugs. Additionally we can reduce the implementation effort.

The logical test-case, i.e. the steps of the test, for granting a discount will look like this:

1. Start the SUT.
2. Select a vehicle.
3. Specify the discount.

4. Check whether the discount has been used for price calculation.
5. Stop the SUT.

The following paragraphs demonstrate how to implement such a scenario.

15.2 Traditional way of implementing data driven testing

The QF-Test definition of a Test-case node says that a Test-case is a workflow together with a specific set of test data. If you have two sets of test data, you have to use two different Test-case nodes. Those nodes can be organized within a Test-set node.

So the conventional way to solve the issue is to implement one Test-case per discount level, like depicted below:

Figure 15.1: Traditional way of data driven testing

Those three nodes are grouped within a Test-set node. The Test-set node also contains the Setup and the Cleanup sequences, which will launch the SUT before each Test-case and will stop the SUT after it. This is to ensure that each test-case has the same pre-conditions. If you do not want to re-start the SUT between the single test runs, you could add a new Test-set to the Test-set 'Discount levels' and move the three Test-case nodes into that Test-set like this:

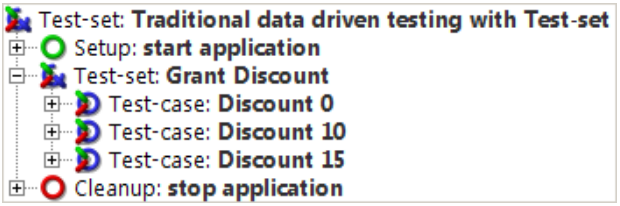

Figure 15.2: Traditional way with a nested Test-set

The chapter Dependencies: Automatically ensuring correct prerequisites for each

Test-case [60 min][174] will show up a more elegant and more efficient way of organizing pre-requisites of test-cases.

As you can imagine this approach could be very exhausting with regards to maintaining test-data, especially if we have to add or remove discount levels. Another disadvantage is that we keep the test-data within QF-Test.

The next section Data driver concept[166] demonstrates how to organize the test-suite to implement the Test-case only once and to keep the test-data separated from the test-case.

15.3 Data driver concept

If we want to run one test-case using different test data sets, we have to define the test data in a data source first. The data source has to be part of a Data driver node. QF-Test offers built-in data source nodes for database tables, CSV files, Excel files and QF-Test data tables. The data is maintained in the test-suite itself. This is the one we will use for the following example. Any other type of data sources, e.g. XML files, can be read by implementing an own script.

Insert a Test-set to the test-suite first. You can choose whatever name you want.

A Data driver node can be inserted into a Test-set via right mouse click and then selecting Insert node→Data drivers→Data driver. You have just to specify a name for this node. The actual data source will then be inserted as child node to that Data driver node. For our example we insert a 'Data table' via a right mouse click at the opened Data driver node and select Insert node→Data drivers→Data table. Now we should detect this dialog:

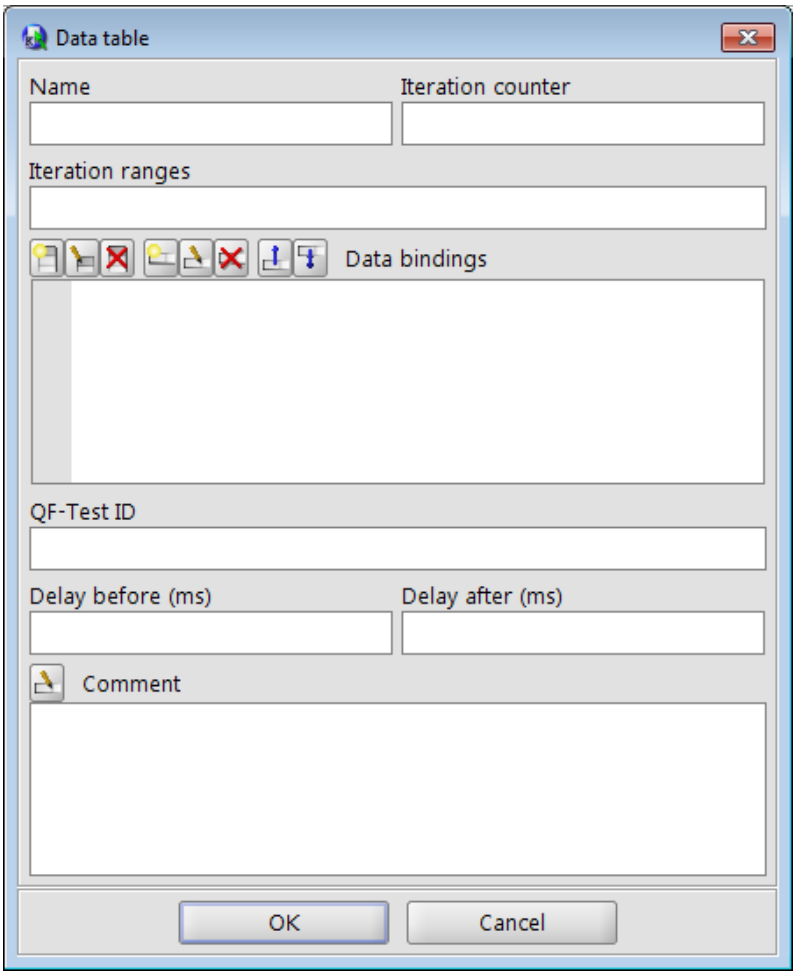

Figure 15.3: Data table dialog

First we have to specify a name for this data source. We should also define a name for the 'Iteration counter' variable. The iteration counter contains the index of the currently executed test data in the test run

The next step is to define the test data. Therefore click on the 'Insert column' button, which is the first button of the 'Data bindings' section. Then you have to define a name for the column, let us set it to 'discount'. After pressing 'OK' you will see that the column has been inserted in the 'Data bindings' area. This column heading will stand for the variable name in the tests later.

Now you can use the 'Insert row' button to insert a new row. Each row will stand for one test data set, i.e. you have to insert three rows by now, where the first row contains 0%, the second 10% and the third 15%.

The table should look like this:

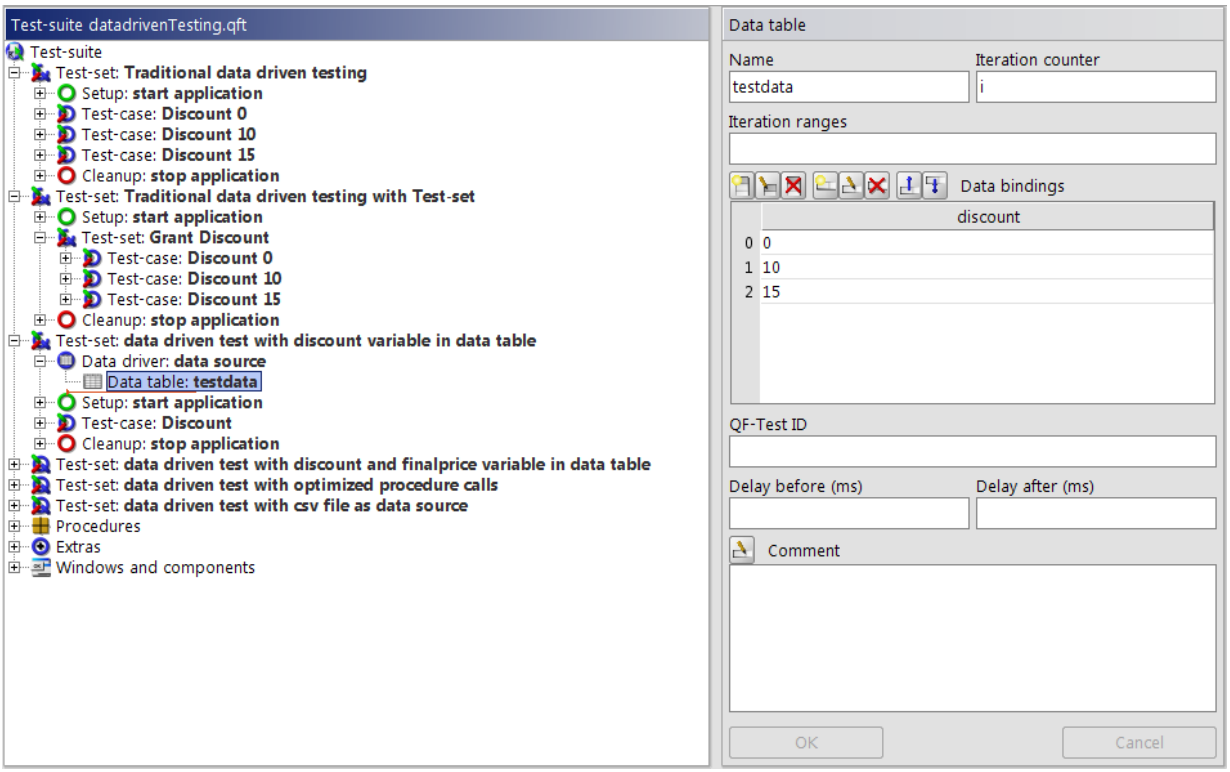

Figure 15.4: The filled data table

The next part is the implementation of the test-case. Therefore we just insert one Test-case node to the Test-set.

Note If you want to insert a Test-case to a Test-set which already contains a Data driver node, you have to select the closed Data driver node to insert it.

The Test-case will contain the procedure calls for the required test steps. The Setup and Cleanup nodes of the previous example can also be copied to the Test-set. The whole Test-set will look like this now:

Test-set: data driven test with discount and finalprice variable in data table
Data driver: data source
Data table: testdata
Setup: start application
Test-case: Discount
Cleanup: stop application

Figure 15.5: Test-set with Data driver

The following step is to make use of the variable 'discount' which has been defined at the 'Data table' node before. We will add that variable as parameter to the 'setDiscount'

procedure call. Once we have done this, we have following result:

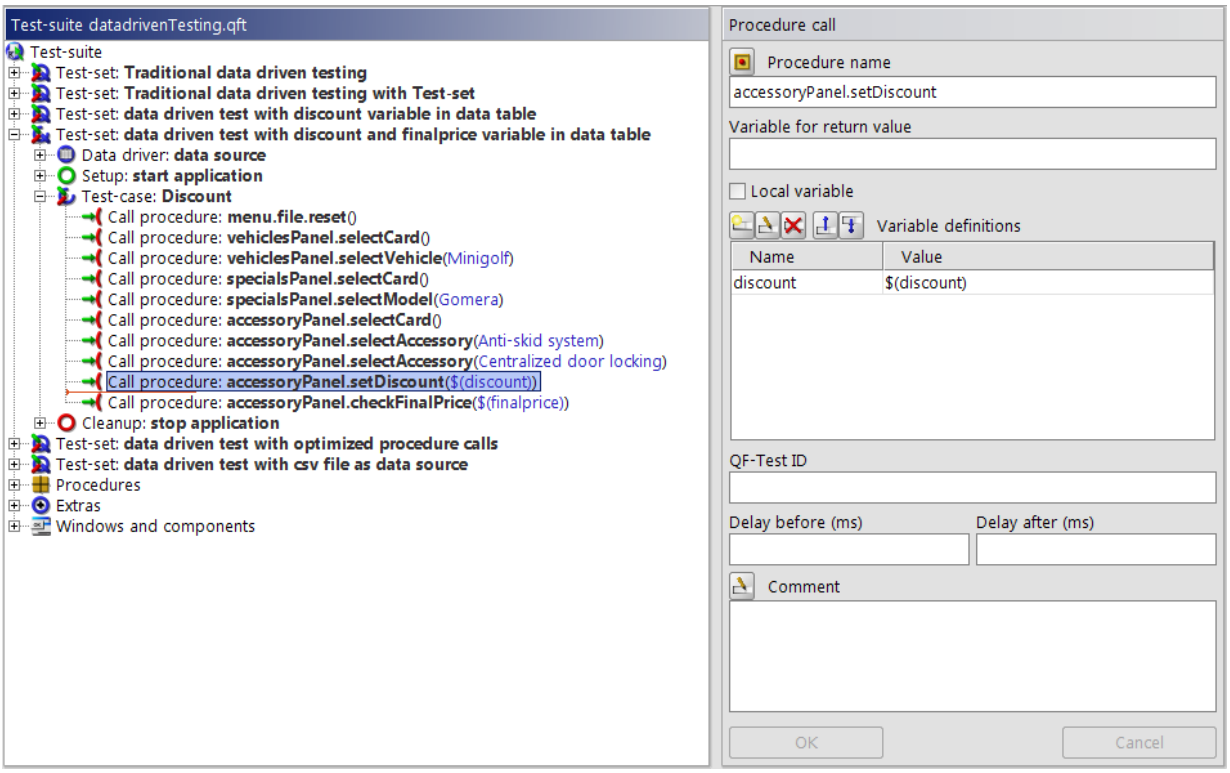

Figure 15.6: Using the `$(discount)` parameter

Now we are ready to launch the Test-set.

After running the tests we should get at least two errors. Those errors come from the different values of the 'Final price' text-field, but our test always verifies the same value. In our case we should also put the expected values for the 'Final price' fields to our test data via adding a second column to the 'Data table' node.

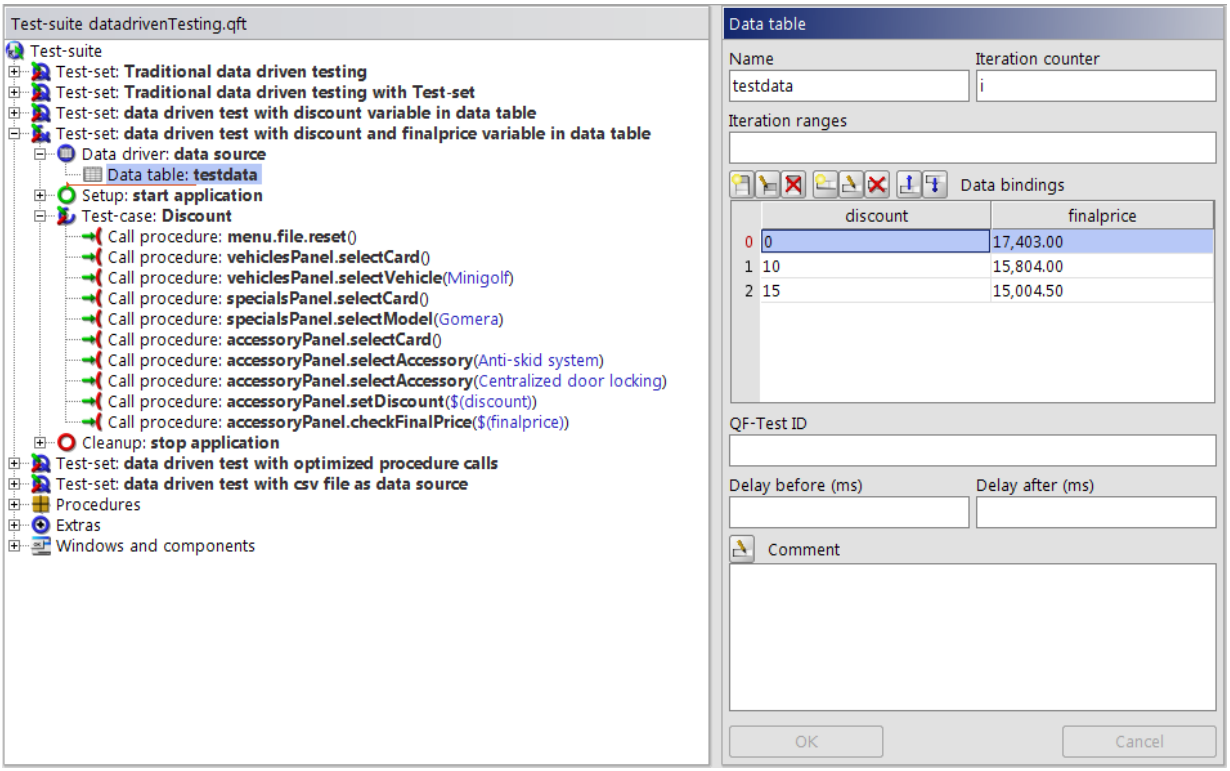

Figure 15.7: Full data table

Another drawback is, that we see the same test-case name in the HTML overview report and in the run-logs for each test run. To avoid this we can edit the 'Name for reports and run-logs' attribute of the Test-case node. In that attribute we have to make use of at least one of the test-case specific data, i.e. in our case 'discount'. So let us set that attribute to 'Discount $(discount)'.

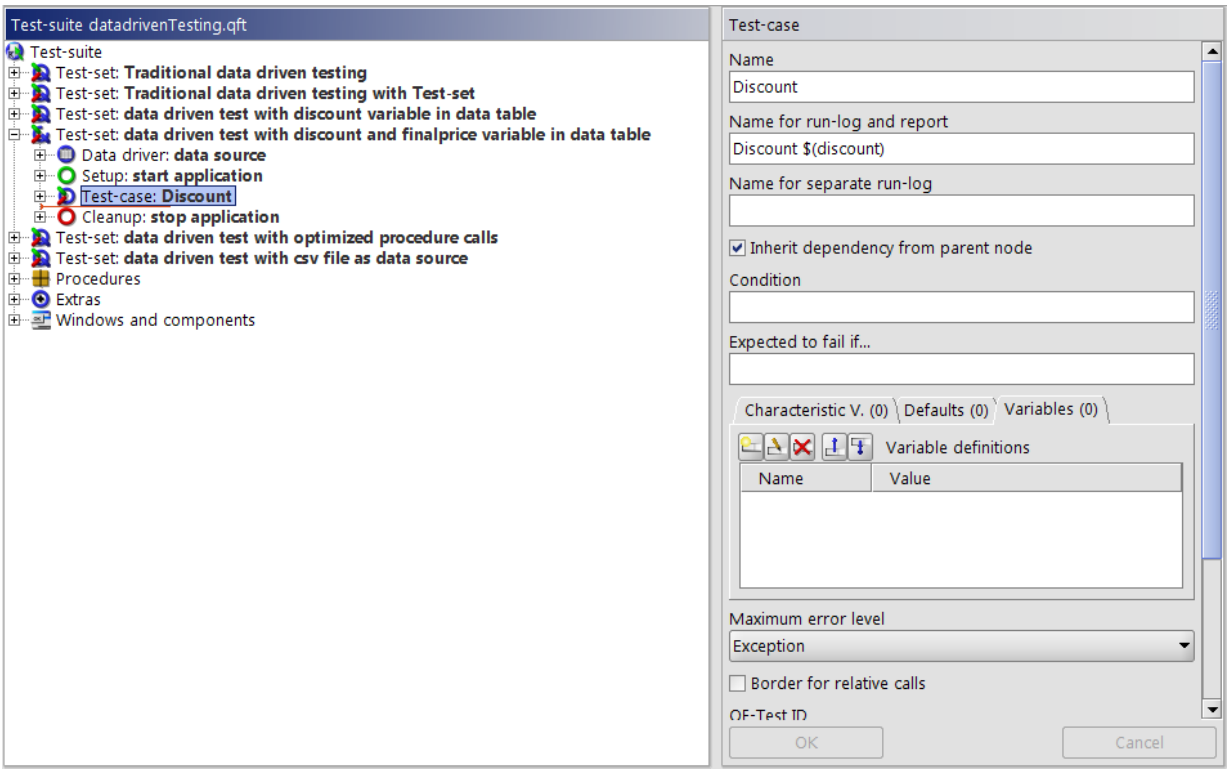

Figure 15.8: Name for run-log and report attribute

If we re-run the tests now, we should get no error anymore and the run-log as well as the HTML report contain three different test-case names. Here you can see the created run-log file:

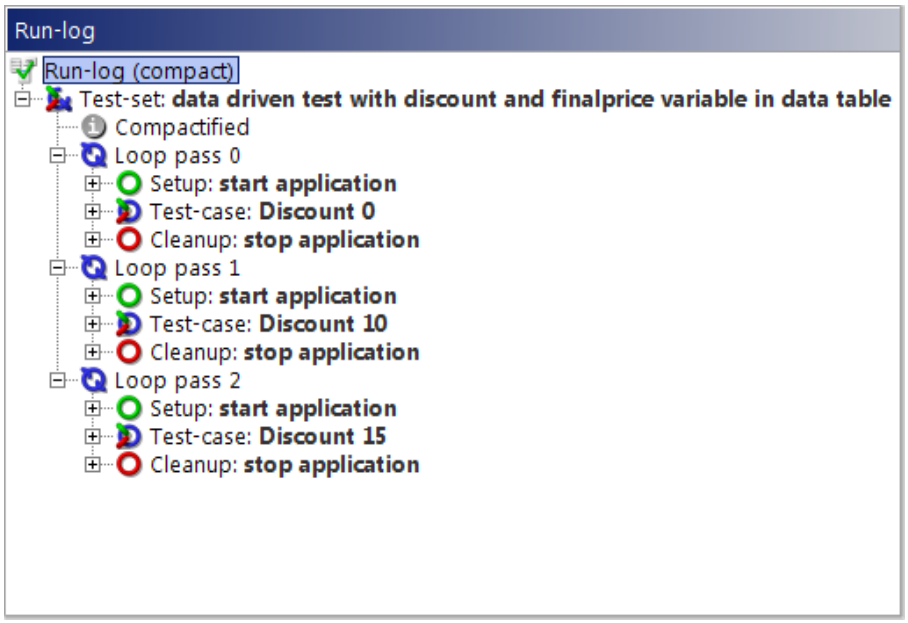

Figure 15.9: Run-log with different names per Test-case

If you want to run one Test-case only without the whole Test-set and its Data driver, it is recommended to set default values for the variables as global ones at Test-suite.

Note If the name of the variable in the Data driver is the same like the name of the procedure, you can also skip the variable definition of the Procedure call. This can be done because the variable of the Data driver will be put at QF-Test's variable stack and so any step within the Test-case can access that variable. You can see this behavior in the demo test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/datadrivenTesting.qft` in the Test-set "data driven tests with optimized procedure calls".

In the demo test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/datadrivenTesting.qft` you can also find an additional Test-set using a CSV file as data source.

15.4 Summary

The Data driver concept of QF-Test allows the user to create logical test-cases and to keep the test data separated from the test flow itself.

It is even possible to use nested Data driver nodes in a Test-case. This can be achieved by creating a Test in a Test-case. This Test can contain the nested Data driver.

You can find a more detailed explanation about data driven testing within QF-Test in the

manual in the chapter Data-driven testing.

The provided test-suite `qftest-4.0.8/doc/tutorial/datadriver.qft` contains more examples for using Data drivers.

Chapter 16

Dependencies: Automatically ensuring correct prerequisites for each Test-case [60 min]

This chapter explains the dependency concept of QF-Test. That concept is important for creating robust test-cases and for defining recovery activities, if a single test-case crashes during the test run. It has been introduced to guarantee that any test-case's prerequisites are fulfilled before running it.

You can find the following examples in the file `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies.qft`. There is also a second test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies_work.qft` to perform your own implementations.

16.1 General

Please open the test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies_work.qft` and take a look at the first Test-set 'Discount Tests'. It contains three different Test-case nodes and the respective Setup and Cleanup to launch and stop the SUT before each Test-case. That is a typical test-suite you might have created yourself.

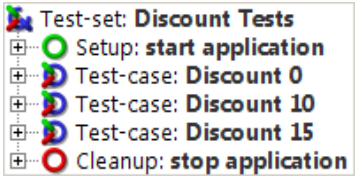

Figure 16.1: First Test-set of dependencies_work.qft

Let us assume you just want to launch one specific Test-case, because that one failed or it verifies a specific defect. If you want to do so, you have either to run the whole Test-set, because you have to ensure that the prerequisites are fulfilled, or you run the Setup manually and run the according Test-case then.

As this is a situation which is very common but not easily solvable with the concepts we know already, QF-Test introduces the Dependencies concept. It is responsible for managing prerequisites of Test-cases and allows you to run one Test-case directly. In that case QF-Test will take care of the prerequisites of the Test-case, e.g. launching the SUT, selecting a vehicle and so on.

A Dependency can contain a Setup, a Cleanup, a Error handler and a Catch node. The Setup of a Dependency will be executed before each Test-case. This is because the correct pre-conditions of a Test-case are a very important aspect of a robust test-run. You could imagine a situation where one Test-case closes the SUT because of a mistake. Then the next test-case has to launch the SUT again. Exactly for this situation, the Dependency concept provides a very stable and fancy solution.

The second aspect of Dependencies is that they optimize the execution of tests. With the current means we have to start the SUT before each Test-case and stop the SUT after each. That is no problem for a small application like the JCarConfigurator, but for huge applications, like Eclipse/RCP application or any ERP system, this might become insufficient. That is why Dependencies call the Cleanup steps only if required.

Another advantage of the Dependencies concept are global Error handler and Catch for recovery management. This feature becomes quite important, if you run lots of test-cases in a series and one is failing and preventing the other test-cases from continuing execution because of modal error dialogs, e.g. 'OutOfMemoryException'.

In a nutshell Dependencies are

1. a place to define the prerequisites of a Test-case.
2. very helpful to make test-cases more independent from other ones.
3. a better approach of implementing Setup and Cleanup steps.
4. allows you to implement recovery steps in case of errors or exceptions.

5. optimizing test-execution.
6. re-usable as they can be defined in the Procedures area.

The following sections demonstrate how to use Dependencies.

16.2 Ensuring prerequisites

Please open the file `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies_work.qft`.

This file contains a Test-set 'Discount Tests' with three Test-cases and the conventional implementation using the Setup and Cleanup. We will change this Test-set to make use of a Dependency now.

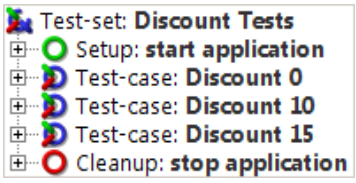

Figure 16.2: First Test-set of dependencies_work.qft

First we have to insert a Dependency node. This can be achieved by a right click at the Test-set and then selecting Insert node→Dependencies→Dependency. Specify a name for that Dependency, e.g. "SUT started".

The next step is to move the Setup and Cleanup nodes into that Dependency. Therefore open the Dependency node and move the mentioned nodes into it. You can do this via drag and drop or via right mouse click Cut and Paste or Ctrl-X and Ctrl-V.

The test-suite should now look like this:

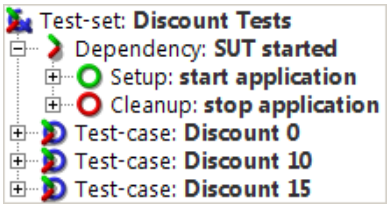

Figure 16.3: Sample test-suite with the first Dependency

Let us test the Dependency now: Please stop all running clients before that, then select one Test-case, e.g. 'Discount 10', and run it.

You should see that the Test-case 'Discount 10' has been executed and the SUT has not been stopped at the end of the test-run. Please open the run-log to make a deeper analysis of what happened.

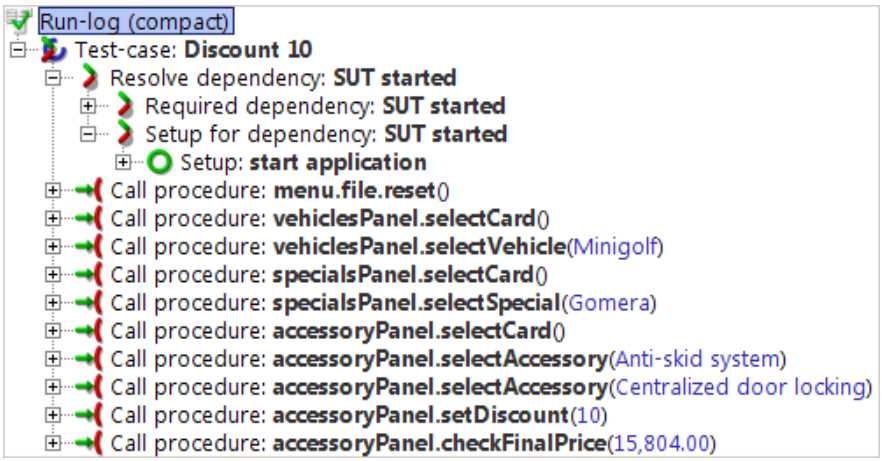

Figure 16.4: The run-log of the execution

If you open the Test-case in the run-log, you can see a 'Resolve dependency' node. Open it and there you find two nodes. The last one shows that the Setup has been executed. The other one will be explained in following chapters.

So far we have determined that the Setup node will be automatically executed before a Test-case, if the Dependency is defined at the Test-set level, but the Cleanup has not been executed so far. If you start another Test-case now, e.g. 'Discount 15', you will get a DuplicateClientExeption at the beginning. QF-Test interrupts the test run and the debugger pops up. Please stop the debugger. Why did that happen?

The Setup node of a Dependency will be executed before each Test-case in any case. That is to ensure that the prerequisites are fulfilled for each single Test-case. The Cleanup node of a Dependency will only be executed on demand, i.e. if the steps of the Setup of that Dependency are not required anymore. In our case the Cleanup steps did not get executed, because both test-cases have the same Dependency. Because of that, we have to change the implementation of the Setup node a little bit. The Setup node should try to determine whether the prerequisites have been fulfilled already. If the pre-conditions are ok, QF-Test does not need to execute the other steps of the Setup, otherwise QF-Test will bring up the required pre-conditions. In our case we have to check, whether the SUT has been launched already. This can be achieved by using a Wait for client to connect node, which waits for the established connection between the SUT and QF-Test. You have to set the 'Timeout' attribute to '0' as the process is expected to be there. If the SUT's process is there, we do not need to start the SUT. If the process is not there, we have to start it. Such actions can be achieved using the try-catch structure, like below:

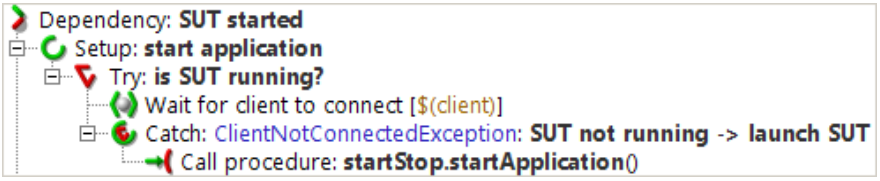
Dependency: SUT started
Setup: start application
Try: is SUT running?
Wait for client to connect [$(client)]
Catch: ClientNotConnectedException: SUT not running -> launch SUT
Call procedure: startStop.startApplication()

Figure 16.5: The new Setup implementation

You could also use the result variable of the Wait for client to connect node and check the result in an If clause.

If we re-run the 'Discount 15' Test-case, it should pass by now.

The next step is to launch the complete Test-set by clicking it and pressing 'Run test'.

All three tests should pass and the SUT should not be stopped in between. We have achieved an optimized test-execution. The Cleanup will not run because all three test-cases refer to the same Dependency. We save a lot of time in executing the test-cases and we are closer to testing a real life behavior as users usually do not restart the SUT all the time.

The next goal is to make that Dependency available for other Test-sets in our project. For this you need to move the Dependency node into the Procedures area. Then click at the Test-set and insert a Dependency reference node via Insert node→Dependencies→Dependency reference. The dialog looks similar to the Procedure call dialog and you can select the Dependency on the same way. The test-suite should now look like this:

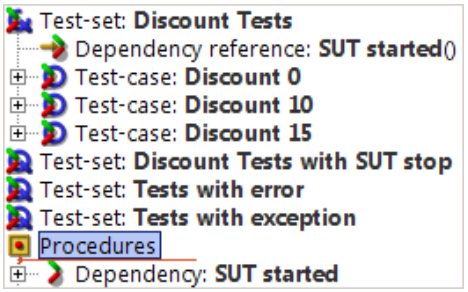

Figure 16.6: The test-suite with a Dependency reference

We recommend to add all Dependencies to a separate Package called 'dependencies'.

If you run the Test-set the first time after moving the Dependency, QF-Test stops the SUT before the first Test-case and re-starts the SUT, because the Dependency from the Test-set is different from the Dependency defined in the Procedures node. But this will be explained more detailed in following chapters.

Please take a closer look at the second Test-set 'Discount Tests with SUT stop' of the demo test-suite. The second Test-case 'Dicount 10' stops the SUT. But the third Test-case also relies on a running SUT. As we know the Dependencies concept already, we should see that the SUT will be started for the third Test-case again. That is exactly the advantage of the Dependencies concept.

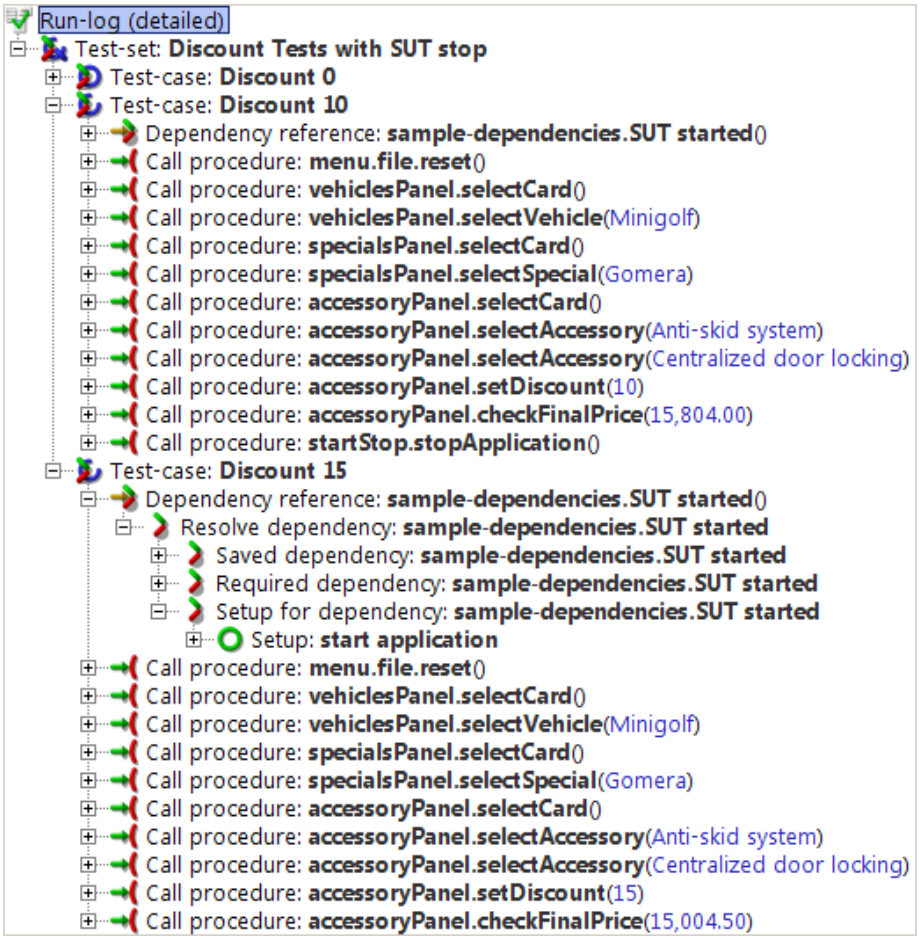

Figure 16.7: Ensuring prerequisites for Test-case 'Discount 15'

16.3 Combining dependencies

Managing prerequisites can be a more complex matter than just verifying whether the SUT is up and running. In many projects several groups of test-cases with different prerequisites like others exist.

Let us assume you want to test a big ERP system with more views, like a 'vendor' view and an 'article' view. Then all tests for the 'vendor' view will rely on an opened 'vendor'

view and all tests for the 'article' view rely on an opened 'article' view. The opening of the specific view again relies on a logged-in user and the login depends on a started SUT. So you can see something like a tree of prerequisites.

QF-Test allows the user to build such a structure of Dependency nodes via adding Dependency reference nodes to a Dependency. We will build up a small example with just two dependencies for the JCarConfigurator now.

In the JCarConfigurator you can open a 'vehicles' dialog via the menu 'Options' -> 'Vehicles'. We want to create certain tests for that dialog now. Later we want to create some tests on the 'accessory' dialog which can also be opened via the menu via 'Options' -> 'Accessories'.

First let us define the tests which we want to create.

Test Case 1: Create vehicle 'test1' with price '100'.

- Start SUT if necessary.
- Open vehicles dialog via the menu.
- Specify 'test1' as name and '100' for the price.
- Press 'New'.
- Close vehicles dialog via pressing 'Ok'.
- Open vehicles dialog via the menu.
- Select the newly created vehicle 'test1'.
- Close the vehicles dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Test Case 2: Create vehicle 'test2' with price '99999'.

- Start SUT if necessary.
- Open vehicles dialog via the menu.
- Specify 'test2' as name and '99999' for the price.
- Press 'New'.
- Close vehicles dialog via pressing 'Ok'.
- Open vehicles dialog via the menu.
- Select the newly created vehicle 'test2'.

- Close the vehicles dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Test Case 3: Create accessory 'testaccessory' with price '12'.

- Start SUT if necessary.
- Open accessories dialog via the menu.
- Specify 'testaccessory' as name and '12' for the price.
- Press 'New'.
- Press 'Ok'.
- Open accessories dialog via the menu.
- Select the newly created accessory 'testaccessory'.
- Close the accessories dialog via pressing 'Cancel'.
- Stop SUT if necessary.

Let us take a closer look at the test-steps of the tests above. We see that each test case requires a running application, so we should implement the 'Start SUT' step as prerequisite. This has to be done in the Setup of a Dependency. The 'Stop SUT' step is an optional step which can be part of the Cleanup of the Dependency. This 'SUT started' Dependency has been implemented by us in the previous examples already, so we could re-use it.

The next issue is that test-case 1 and test-case 2 require an opened vehicle dialog. Because we plan more tests in that area, we can create a new Dependency 'vehicles dialog opened', which will contain the opening of the dialog as Setup and the closing of the dialog via 'Cancel' as Cleanup. We are able to open this dialog only if the SUT is up and running already, so this Dependency is dependent on the 'SUT started' Dependency. The implementation of that 'open vehicle dialog' Dependency looks like this:

Dependency: vehicles dialog opened
Dependency reference: SUT started()
Setup: open dialog
Try: is dialog opened?
Wait for component [VehiclesDialog=>$(client)]
Catch: ComponentNotFoundException: dialog not open -> open it
Call procedure: menu.options.open vehicles dialog()
Cleanup: close dialog
Try: dialog already closed?
Wait for component absence [VehiclesDialog=>$(client)]
Catch: ComponentFoundException: close dialog
Call procedure: vehiclesDialog.pressCancel()

Figure 16.8: 'vehicles dialog opened' Dependency

Note

In the Setup we have to check whether the dialog is already opened, because a previous test-case could have forgotten to close the dialog. The 'Timeout' attribute of the Wait for component to appear node is set to '0' because the dialog is expected to be here. If the dialog is already opened, it is fine and we can continue, otherwise we have to open the dialog.

We have to create a Dependency 'accessories dialog opened' as well. This Dependency is similar to the 'vehicles dialog opened' Dependency, but it deals with the accessory dialog instead of the vehicles dialog.

After creating those dependencies we have to record the according test-steps and create the test-cases. The test-steps have already been implemented by us and you can find them in the according Package structure in the test-suite.

The test-cases should be organized in one Test-set called 'Combined Dependencies Tests'. This Test-set should contain two more Test-sets. The first one is 'Tests for vehicles' and the second one is 'Tests for accessories'. The Test-set 'Tests for vehicles' has to contain the implementation of test-case 1 and test-case 2 and a Dependency reference to the 'vehicles dialog opened' Dependency. The second Test-set 'Tests for accessories' has to contain test-case 3 and a Dependency reference to the Dependency 'accessories dialog opened'.

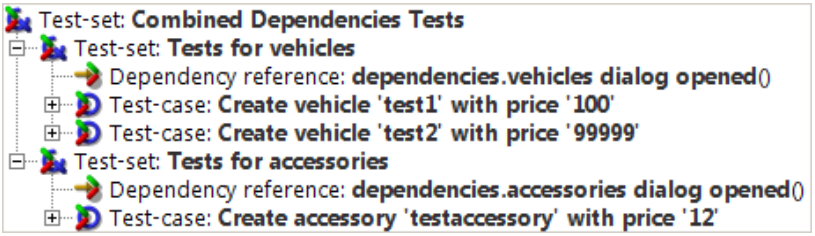
Test-set: Combined Dependencies Tests
Test-set: Tests for vehicles
Dependency reference: dependencies.vehicles dialog opened()
Test-case: Create vehicle 'test1' with price '100'
Test-case: Create vehicle 'test2' with price '99999'
Test-set: Tests for accessories
Dependency reference: dependencies.accessories dialog opened()
Test-case: Create accessory 'testaccessory' with price '12'

Figure 16.9: Implementation of specified test-cases

If you run that Test-set now, you will see that QF-Test stops the SUT first, because that comes from a dependency of the previous examples. Then QF-Test starts the SUT, performs the steps of test-case 1 and then the steps of test-case 2. Then it will perform the steps of test-case 3. If you take a closer look at the beginning of test-case 3 in the run-log, you will see that the Cleanup of the 'vehicles dialog opened' Dependency has been executed. That happened because the 'vehicles dialog opened' Dependency was not required anymore. The following test-case test-case 3 required the 'accessories dialog opened' Dependency. However, both Dependencies require the 'SUT started' Dependency, that is why that Cleanup of the 'SUT started' Dependency has not been executed.

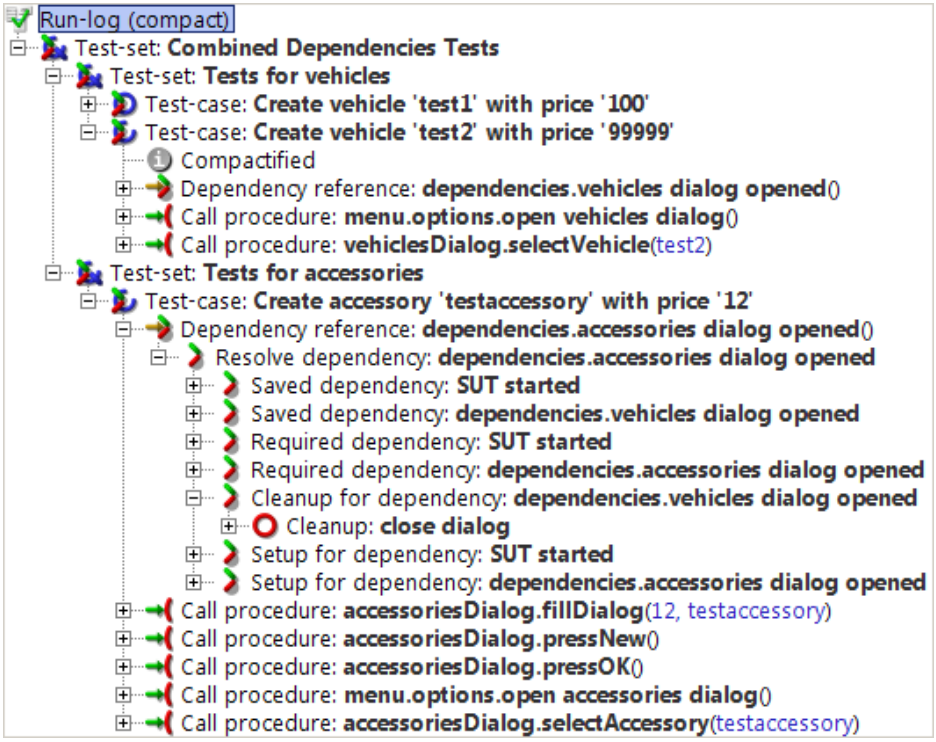
Run-log (compact)
Test-set: Combined Dependencies Tests
Test-set: Tests for vehicles
Test-case: Create vehicle 'test1' with price '100'
Test-case: Create vehicle 'test2' with price '99999'
Compactified
Dependency reference: dependencies.vehicles dialog opened()
Call procedure: menu.options.open vehicles dialog()
Call procedure: vehiclesDialog.selectVehicle(test2)
Test-set: Tests for accessories
Test-case: Create accessory 'testaccessory' with price '12'
Dependency reference: dependencies.accessories dialog opened()
Resolve dependency: dependencies.accessories dialog opened
Saved dependency: SUT started
Saved dependency: dependencies.vehicles dialog opened
Required dependency: SUT started
Required dependency: dependencies.accessories dialog opened
Cleanup for dependency: dependencies.vehicles dialog opened
Cleanup: close dialog
Setup for dependency: SUT started
Setup for dependency: dependencies.accessories dialog opened
Call procedure: accessoriesDialog.fillDialog(12, testaccessory)
Call procedure: accessoriesDialog.pressNew()
Call procedure: accessoriesDialog.pressOK()
Call procedure: menu.options.open accessories dialog()
Call procedure: accessoriesDialog.selectAccessory(testaccessory)

Figure 16.10: Run-log of nested Dependencies

The ability to combine dependencies or to call the Cleanup of dependencies on demand can enable you to put a lot of preparing and cleaning steps into a Dependency. Another use-case for the JCarConfigurator could be to create a 'Vehicle created' Dependency, which creates a vehicle, which is required for the test and removes it afterwards. That would be an approach to ensure that the test-data has been established correctly.

16.4 Error and exception handling

16.4.1 Error handling

In the test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies_work.qft` you can find a Test-set 'Tests with error'. The second Test-case of that Test-set fails.

Figure 16.11: Test-suite of Error handler

Let us assume that we want to trigger a specific behavior after test-cases, but only if it failed. In our case we could stop the SUT to guarantee that the following test-case can rely on a clean environment again. We know that the Setup node is executed before each Test-case and the Cleanup node will only run on demand. So how could we run specific steps only in case of an error?

The solution is the so called Error handler node for a Dependency. You have to click at the closed Cleanup node of that Dependency and insert the Error handler node via right mouse click and selecting Insert node→Dependencies→Error handler.

In the Error handler you can define the steps for stopping the SUT again. The Dependency `SUT started` should look like this now:

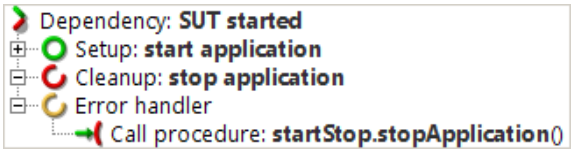

Figure 16.12: Dependency with Error handler

Please run the whole Test-set 'Tests with error' and switch into the run-log after the test-run has finished.

In the run-log you can see that the Error handler was executed after the second Test-case only.

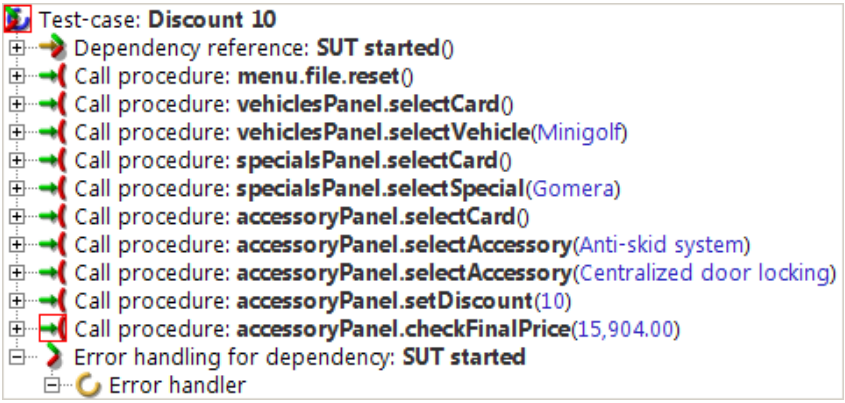

Figure 16.13: Run-log for Dependency with Error handler

16.4.2 Exception handling

In the previous section we have learned that Error handler nodes contain steps which are executed if the Test-case fails. Besides errors also exceptions can occur during a test-run. An exception is an unexpected behavior during the test-run, e.g. an error dialog appears suddenly or a component could not be found anymore. How should we handle such exceptions?

You can find an example Test-set 'Tests with exception' in the test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/dependencies_work.qft`.

Of course you could surround the test-steps in each Test-case by try-catch and implement a dedicated exception handling in each single Test-case, like implemented in the demo Test-set. But this approach could lead to a lot of redundancy and makes the Test-cases a little bit more unreadable.

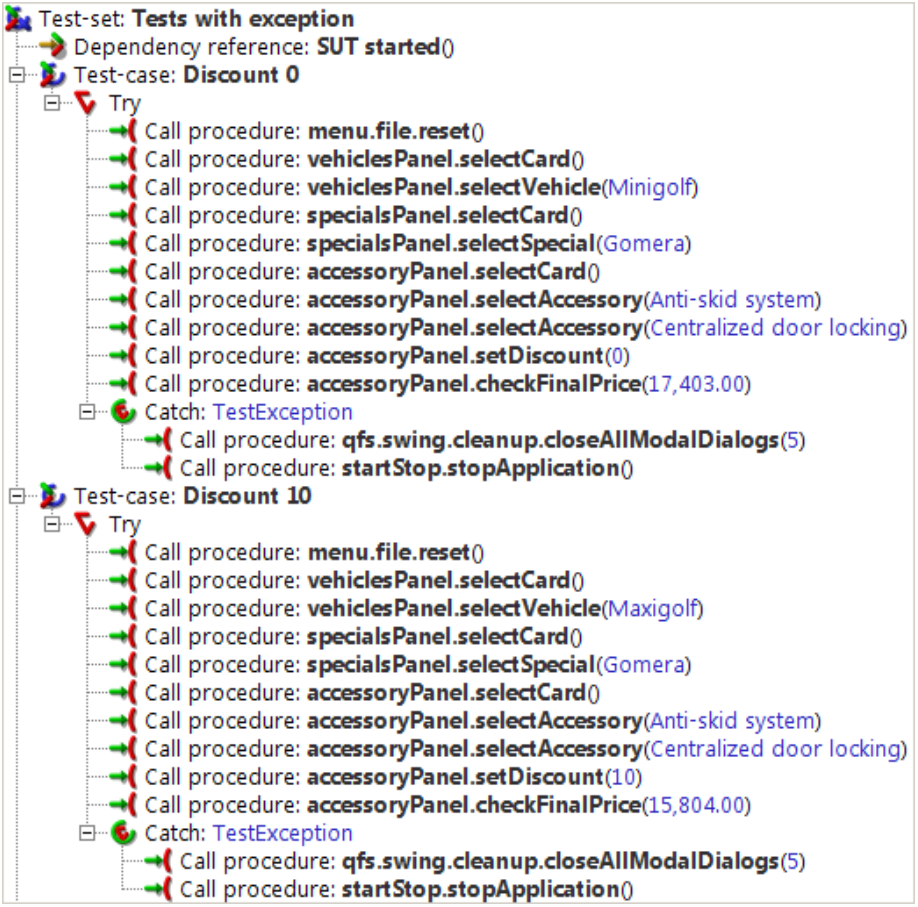

Figure 16.14: Try-catch nodes in each Test-case

Our goal is to reduce the redundancy and to move the exception handling - which is always the same - to one global location. This location will be our Dependency.

The first step is adding the Catch node to the Dependency via clicking at the closed Error handler node and selecting Insert node→Control structures→Catch in the menu. Then we can copy the procedure-calls for stopping the SUT into that Catch. Now the try-catch nodes in the single test-cases become unnecessary. Please move the procedure-calls out of the Try and remove the empty try-catch structure.

Your test-suite should look like this now:

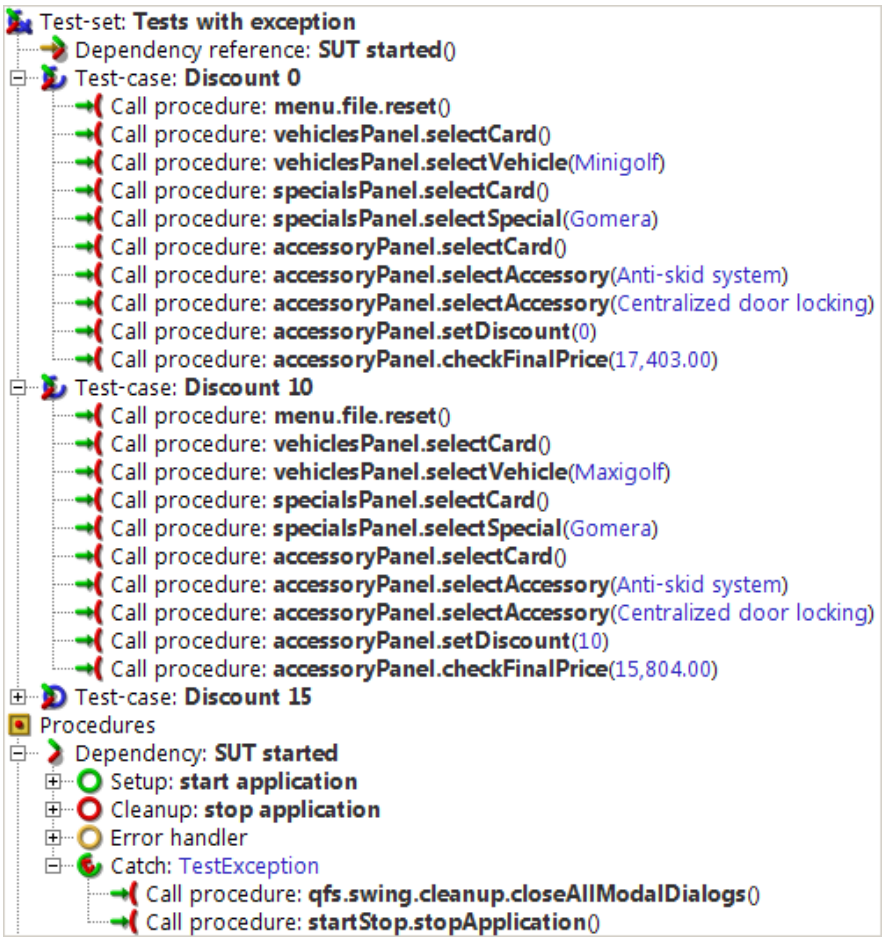

Figure 16.15: Test-suite with Catch

Now you can start the Test-set 'Test with exception'. The second Test-case throws an IndexNotFoundException, because the selected vehicle does not exist. It is intended to handle that exception in the Catch of the Dependency.

Note

If the debugger is activated, QF-Test interrupts the test-run at the place where the exception arises. In this case you might have to re-throw the exception via the 'Rethrow exception' button. An alternative is to deactivate the debugger via Debugger→Enable debugger.

After the test-run you can open the run-log and verify what happened.

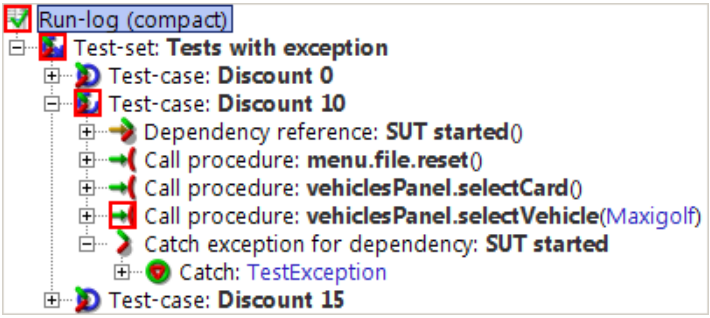

Figure 16.16: Run-log of a Dependency with Catch

In a normal project you should at least try to catch a 'TestException' and the Catch should contain a procedure call of either `qfs.swing.cleanup.closeAllModalDialogs` or `qfs.swt.cleanup.closeAllModalDialogsAndModalShells`. Those procedures close all modal dialogs, e.g. dialog windows, which are blocking the execution.

16.4.3 Summary

You have seen that you can implement a very strong recovery system for your test-cases using the Error handler and, even more important, the Catch of Dependencies.

In most projects the global Catch node becomes very important, especially if ComponentNotFoundExceptions and ModalDialogExceptions appear.

16.5 More about dependencies

In the previous sections we have learned that we can combine several dependencies and that the Cleanup of a Dependency is only executed, if the stack of combined dependencies changes. We could also instruct a Dependency to call its Cleanup every time after a Test-case. This can be achieved by checking the attribute 'Forced cleanup' of the Dependency.

There are even more interesting features about dependencies, e.g. you can use a variable instructing QF-Test to run the Cleanup. This variable is called 'Characteristic variable'. You will find more details about that feature in the manual Dependencies. This approach could be used to create one 'user login' Dependency which will only run through its Cleanup, i.e. perform the logout, if another user is required by the next Test-case.

A detailed description of dependencies can be found in the manual at chapter Dependencies.

Chapter 17

Automated creation of basic procedures [60 min]

This chapter describes how procedures for each GUI component can be created automatically by QF-Test. The advantage of this approach is that you do not need to record each step of your tests manually. Furthermore you will also get a standardized structure of Packages and Procedures for testing all dialogs of your SUT.

You can find the following examples in the file `qftest-4.0.8/doc/tutorial/advanced-demos/en/automated_procedures.qft` . There is also a second test-suite `qftest-4.0.8/doc/tutorial/advanced-demos/en/automated_procedures_work.qft` to perform your own implementations.

17.1 General

If we want to create tests for all features of the JCarConfigurator, we have to record steps on each element that will be touched by a test. The JCarConfigurator is a small application with five dialogs and perhaps approximately thirty graphical elements. So creating all important test-cases for this application can be achieved within one or two days. But imagine a big project like an ERP system with about fifty dialogs and hundreds of elements. Creating test-cases for such a huge system will definitely take longer and also the maintenance of recorded tests could become quite difficult.

As first organizational step we recommend to record each test step as a Procedure and call it from the Test-cases requiring it. If you organize your tests in different test-suites, you could split those test-suites into two levels. The first level contains only GUI component related procedures and the second level contains only test-cases calling procedures of level one.

The approach of implementing each test step as procedure brings us to a situation where we could split the work into two parts:

1. Creating and maintaining the procedures representing the test steps
2. Creating and maintaining the test-cases

QF-Test brings a feature which creates that basic procedures for each GUI component automatically. The usage of this feature should drastically minimize the time for creating test-suites and test-cases and supports you in making easier maintainable test-suites.

You can find a demo suite containing some test-cases created by this `qftest-4.0.8/doc/tutorial/advanced-demos/en/automated_procedures.qft`.

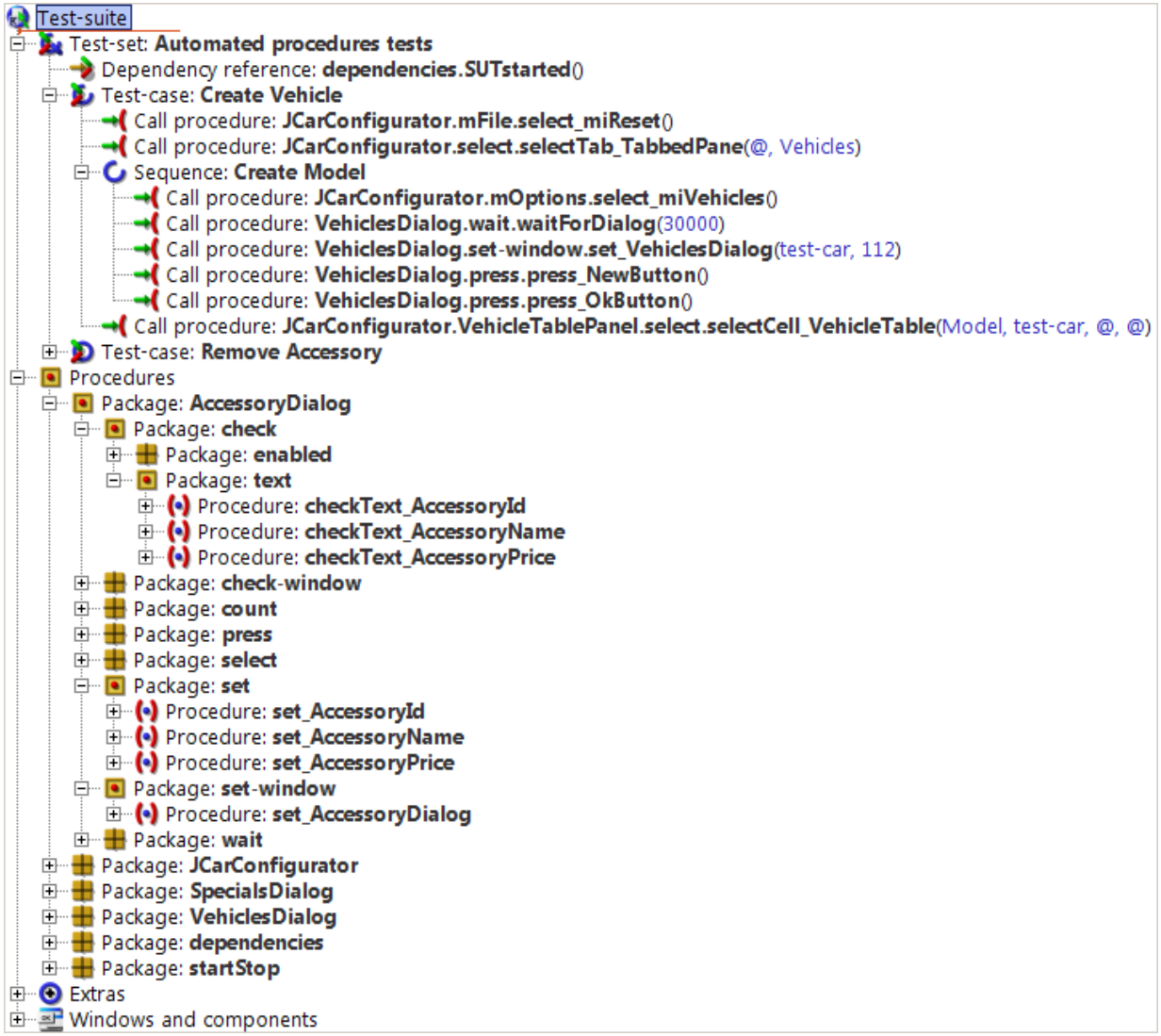

Figure 17.1: Screenshot of test-suite

The following sections demonstrate how to create those procedures and organize your test-cases.

17.2 Automated creation of procedures

Please open the file `qftest-4.0.8/doc/tutorial/advanced-demos/en/automated_procedures_work.qft`

This test-suite contains a Test-set referring to a 'Start SUT' Dependency.

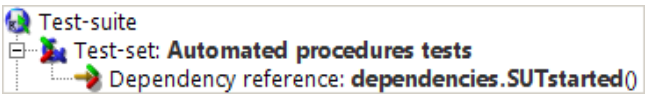

Figure 17.2: The test-suite automated_procedures_work.qft

First we have to launch the SUT via selecting the Dependency reference and pressing 'Start test run'.

Once the SUT is up and running we are ready to record the test steps. Normally we would press the 'Start recording' button, then record the according test steps and then press 'Stop recording'. After that we would re-organize the recorded steps, i.e. generating procedures and parametrizing them. Exactly those steps can be performed automatically now.

Let us generate the basic procedures for the main window first. Before we can start, please modify the settings of QF-Test.

Open the QF-Test options via Edit→Options. Then switch to 'Record' -> 'Procedures'. Please change the value for the 'Configuration file for recorded procedures' to the full path of the demo configuration test-suite at `qftest-4.0.8/demo/procbuilder/carconfig-procbuilderdef.qft`. Then close the 'Options' via pressing 'OK'. You will see more details about that file in the following chapters.

After altering the settings proceed with following steps:

- Press the 'Record procedures' button.
- Perform a right mouse-click in the SUT.
- Select 'Whole window'.
- Press the 'Record procedures' button again.

Now QF-Test creates the basic procedures of the main window. You should see a new Package called `procbuilder` under Procedures of your test-suite. This package contains several packages and procedures for actions on the whole dialog and its single GUI components.

Note The current configuration created a Package `JCarConfigurator` under the `procbuilder` Package. This Package is indicated to be the container for all actions performed on the main window.

Figure 17.3: The recorded procedures

Please note that the involved components have also been recorded in Windows and components.

The next step is to check which procedures are useful for us and which are obsolete.

Let us take a closer look at the created Packages:

Package	Content
`JCarConfigurator`	This package contains all procedures for actions on components of the JCar-Configurator window. It has been created because the configuration file uses the component hierarchy for creating the package structure.

Table 17.1

The Package `JCarConfigurator` contains several packages which are:

Package	Content
`MenuBar`	This package contains procedures for clicking at the items of the menu.
`VehicleTablePanel`	This package contains all procedures for actions on components on the 'VehicleTable' panel of the SUT. In our case there are only procedures for the 'VehicleTable' object, because that is the only component on that panel.
`check`	This package contains procedures for checking components.
`check-window`	This package contains procedures which can be called to check the components of one frame using one call. Those procedures are container procedures.
`get`	This package contains procedures for getting the current values of components, e.g. reading the text of a text-field.
`select`	This package contains procedures for selecting items of elements - in our case only one procedure to select a tab in the TabbedPane.
`set`	This package contains procedures for setting several components of the current window - in our case several setters for the text fields of the frame.
`set-window`	This package contains procedures setting several component of certain frames or dialogs - in our case one procedure calling all procedure for the package 'set' of the JCarConfigurator frame. Those container procedures are typical workflow procedures.
`wait`	This package contains all procedure waiting for several components.

Table 17.2

In our case all procedures might be useful. So we can move the whole 'JCarConfigurator' Package from the `procbuilder` Package directly under the Procedures node. Press 'Yes' on the occurring 'Update Reference' dialog.

Now the procedures can be used by test-cases or other procedures.

Finally our test-suite should look like this:

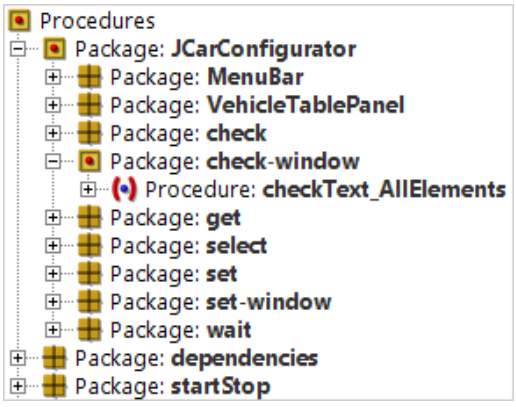

Figure 17.4: The test-suite containing the procedures

Repeat that recording now for the 'Specials' and the 'Accessories' panel and just move the respective procedures or packages for the important components to the 'JCarConfigurator' package. In our case these are only the packages 'SpecialsPanel' and 'AccessoryTablePanel'.

The full test-suite should then look like this:

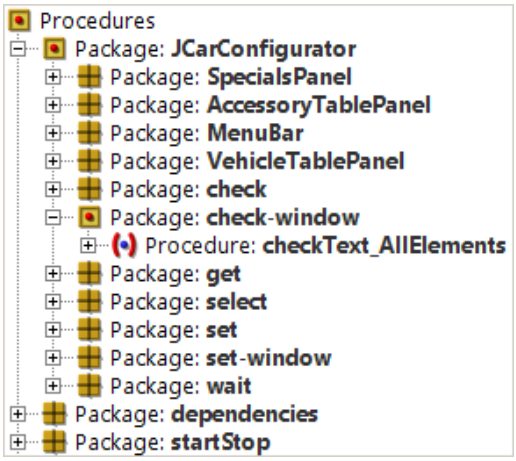

Figure 17.5: The procedures for all panels

We are ready for creating test-cases using the automatically recorded test steps. You could of course also repeat that recording for all other dialogs, e.g. the 'Vehicles' dialog which can be reached via the menu 'Options' -> 'Vehicles' as well as for the 'Accessories' and 'Specials' dialogs.

You do not need to record the whole window all the time. You could also select 'Component only' to record one dedicated component or 'Component with children' to record procedures for a certain panel.

17.3 Configuration of the automated creation

17.3.1 Introduction

In the previous example we used the file `qftest-4.0.8/demo/procbuilder/carconfig-procbuilderdef.qft` as configuration file for the automated creation. In this section we want to take a closer look at the configuration capabilities of QF-Test. So please open that file.

Figure 17.6: The current configuration

The package `procbuilder` is the root-package for all created packages. If you want to use another name for the automatically created procedures, feel free to rename the package.

If you open that package, you can see the 'class' level. This level describes the classes of the GUI components which should be taken into account for creating the packages. The next levels contains information about the created package structure and procedures. You can find a detailed description of those capabilities in the manual in chapter The Procbuilder definition file.

17.3.2 First example

In our first example we want to create a new settings file which we will build up step by step.

Please perform following actions at the beginning:

- Open a new test-suite file and save it. Specify a name like 'mySettings.qft' or something similar.
- Create a new Package with the name 'myProcedures'.

The new test-suite should look like this:

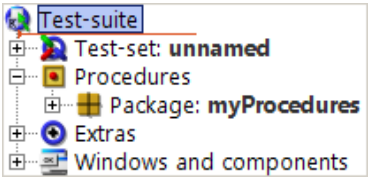

Figure 17.7: The own configuration file

Now we are ready to configure procedure templates for specific classes. Let us create procedures for the text-fields on the main panel of the JCarConfigurator first. It might be interesting for our project to check the content of those text-fields, so we need to create procedures for checking the text of each text-field.

To create those procedures we have to insert a new package to the package 'myProcedures'. Please call that package 'javax_swing_JTextField'. 'javax.swing.JTextField' is the class of those text-fields, but we have to use '_' in the package's name, because '.' is not allowed in that attribute. This package will instruct QF-Test to create the procedures in it, if it meets a component of the class 'javax.swing.JTextField' only. That is an important aspect, because we should realize that procedure templates can be created per classes of components.

Consecutively we want to create a template for a procedure checking the text of a component. The template procedure has to be part of another package. The name of this package should indicate its purpose, e.g. 'checkers'. After creating the 'checkers' package you can add a procedure 'checkText' to it. The procedure should contain a 'Check Text' node which is meant to check the text of components. Please insert such a 'Check text' node via right mouse click at the expanded procedure and Insert node→Check nodes→Check Text. Specify `$(client)` as client, 'dummy' as QF-Test component ID and `$(text)` as text.

Of course, after confirming these entries, we will get a warning that a component named 'dummy' does not exist. In this case we are allowed to ignore this warning.

The test-suite will then look like this:

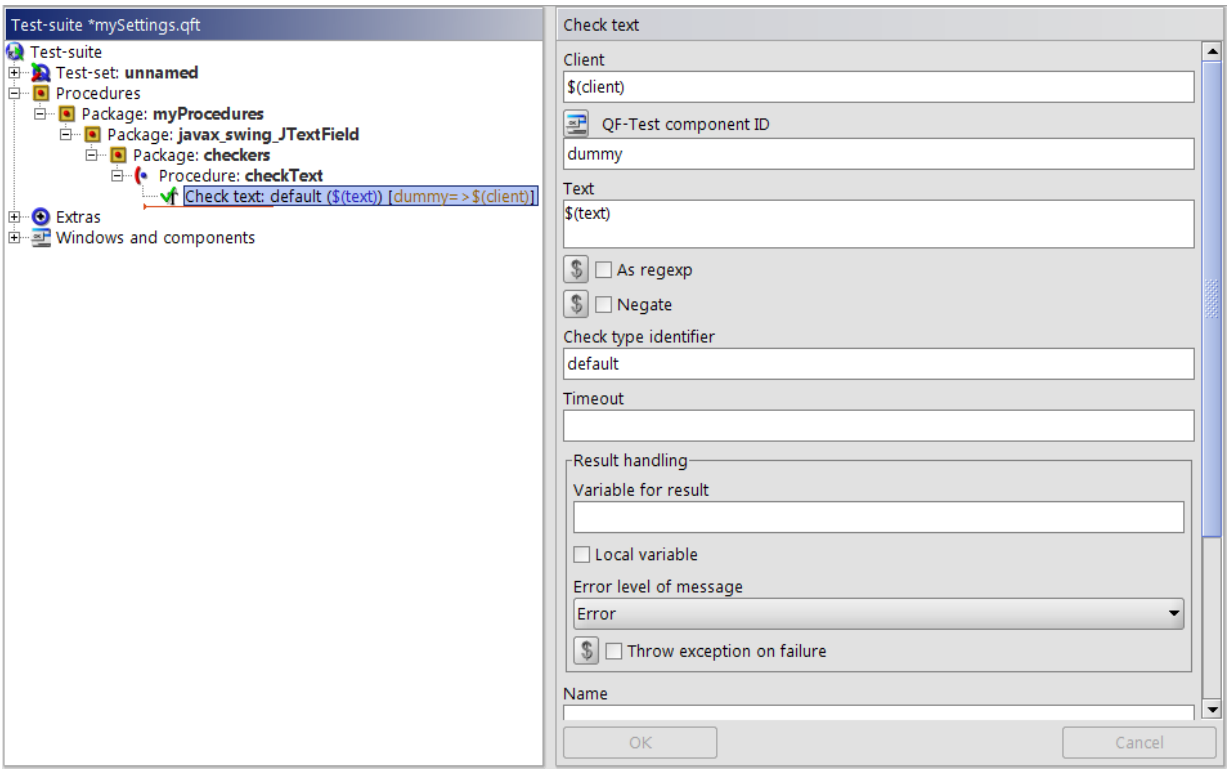

Figure 17.8: The `checkText` procedure

The procedure contains an empty default value for the parameter ’text’.

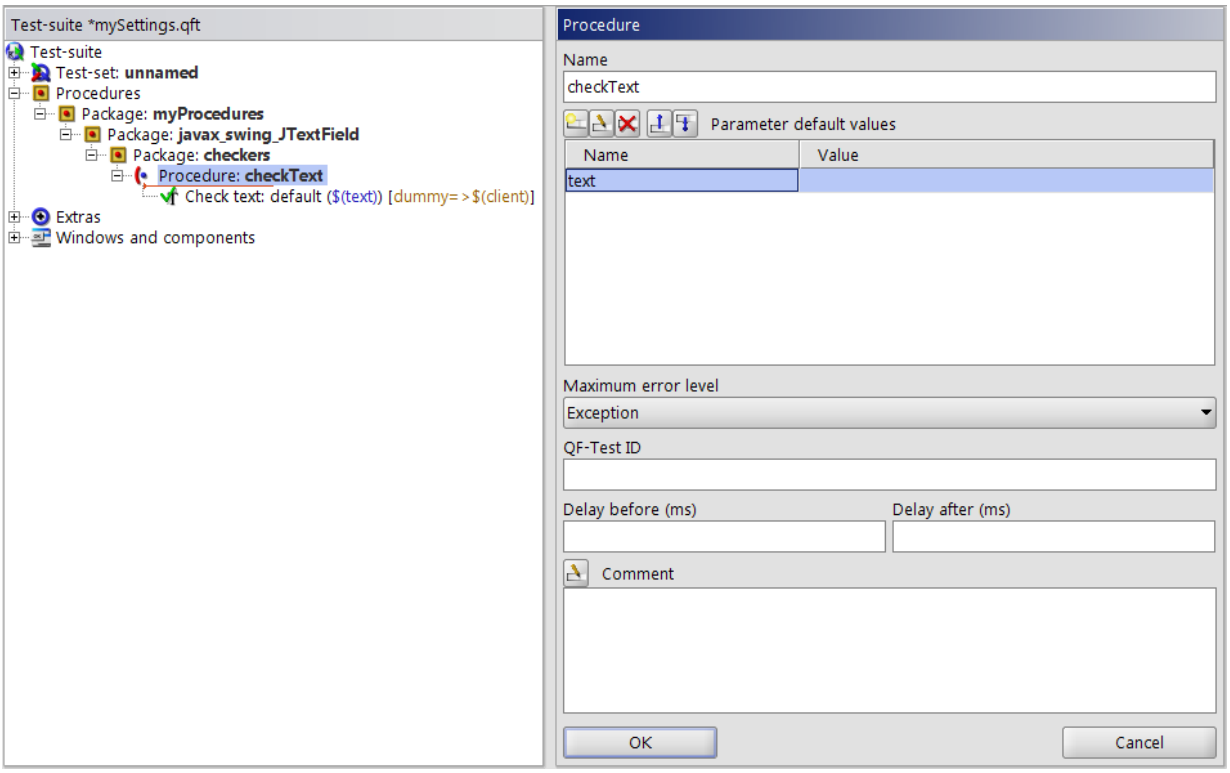

Figure 17.9: The `checkText` procedure with parameters

Now we have nearly completed our first procedure template, but we have to think about one more issue. Each component has its specific and unique QF-Test ID, so it would be great to use that QF-Test ID attribute immediately during creation, otherwise we have to do that manually afterwards. We also want the procedure to have a component-related name instead of 'checkText' only. The place holder `<COMPID>` points out to QF-Test to replace it with the QF-Test ID of the current component. So we have to change the procedure name to `checkText_<COMPID>` and we have to insert the `<COMPID>` place holder directly in the QF-Test component ID attribute of the 'Check text' node.

Finally the template procedure should look like this:

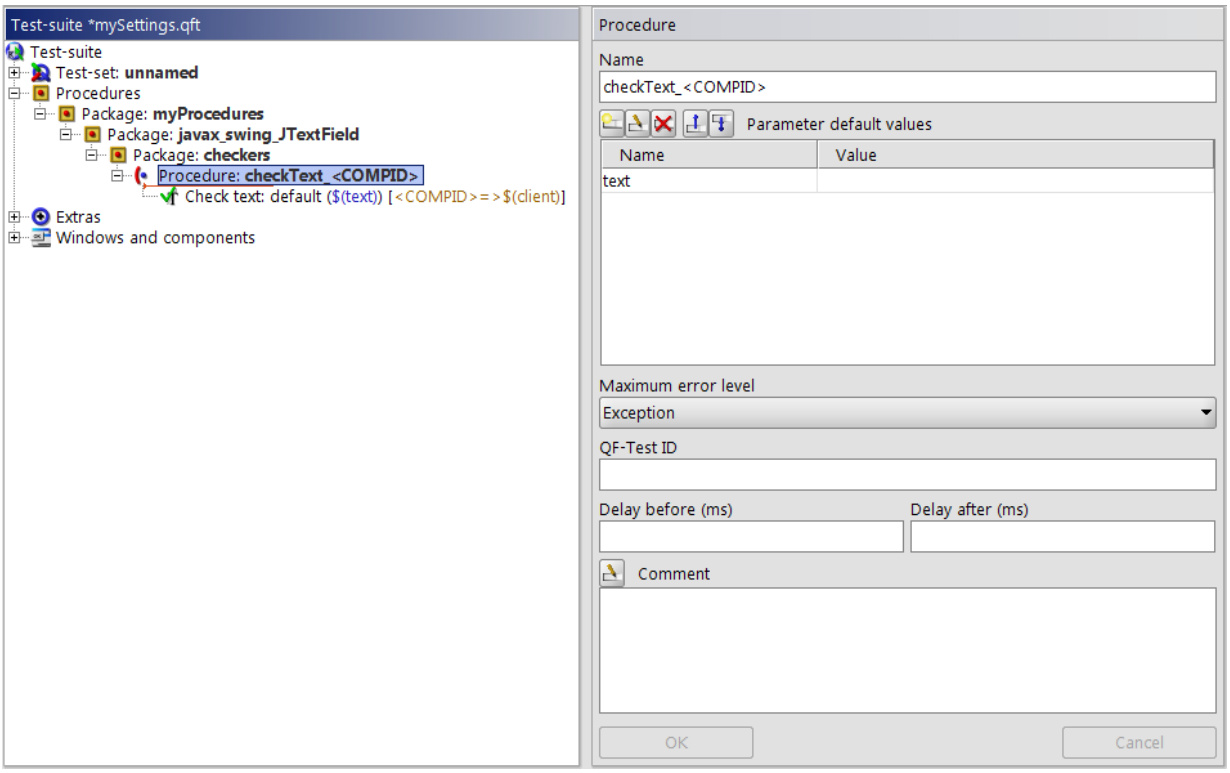

Figure 17.10: Using the `<COMPID>` place holder

We are ready to begin now. First we have to order QF-Test to use that file. That is achieved via the options of QF-Test. Open the options dialog via Edit→Options and change to 'Record' -> 'Procedures.' Specify the path of your 'mySettings.qft' for the 'Configuration file for recorded procedures' and press 'OK'.

Then start the JCarConfigurator again. Once it is up and running, please proceed with following steps:

- Press the 'Record procedures' button.
- Perform a right mouse-click in the SUT.
- Select 'Whole window'.
- Press the 'Record procedures' button again.

Congratulations, you have instructed QF-Test to record test steps for you.

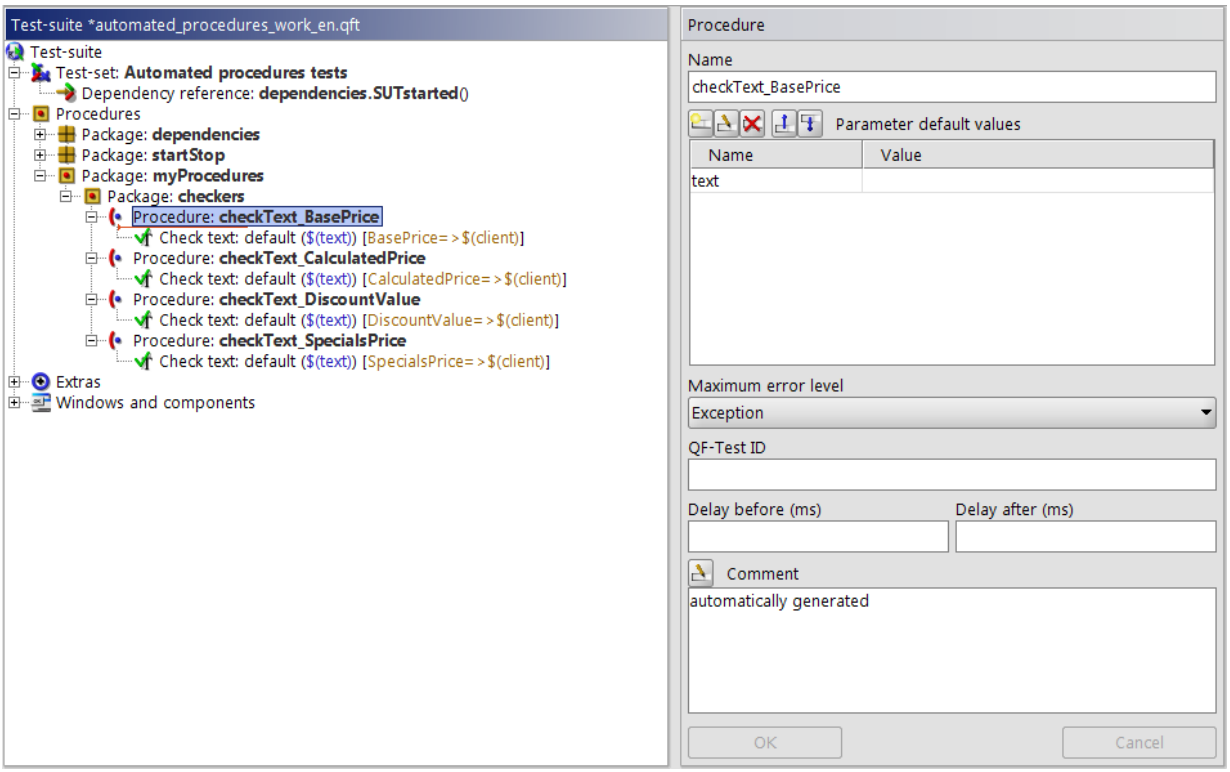

Figure 17.11: Your first automatically created procedures

17.3.3 Using the current text for checking

The 'checkText' procedures have a parameter 'text' for the text to check. Up to now we have to specify that value each time we call that Procedure of the according text-field. Let us assume we have a scenario where we intend to check the default values of the JCarConfigurator after its startup. In this case we would have to add the single procedure calls of all four 'checkText' procedures and specify the according values. However, QF-Test offers a place holder to set the current value of a text-field automatically to the created procedures. Therefore we have to change the default-value of the 'text' parameter to `<CURRENTVALUE>` in our configuration test-suite. After that you have to ensure that the `myProcedures` package doesn't exist under the Procedures node anymore, then re-create the packages like in the previous example. Otherwise QF-Test will create a package `myProcedures1` to ensure that each creation process has its own unique target package.

The configuration file looks like this:

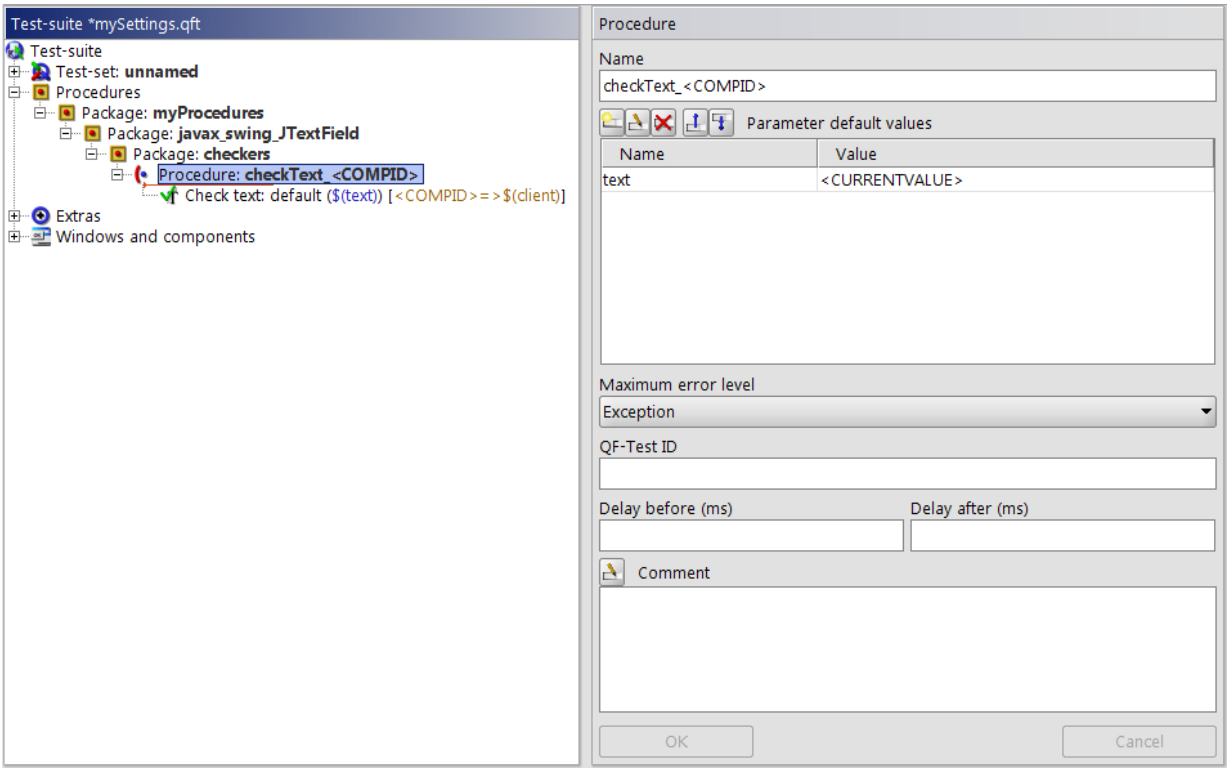

Figure 17.12: The configuration for the current text

The newly created procedures displays like below:

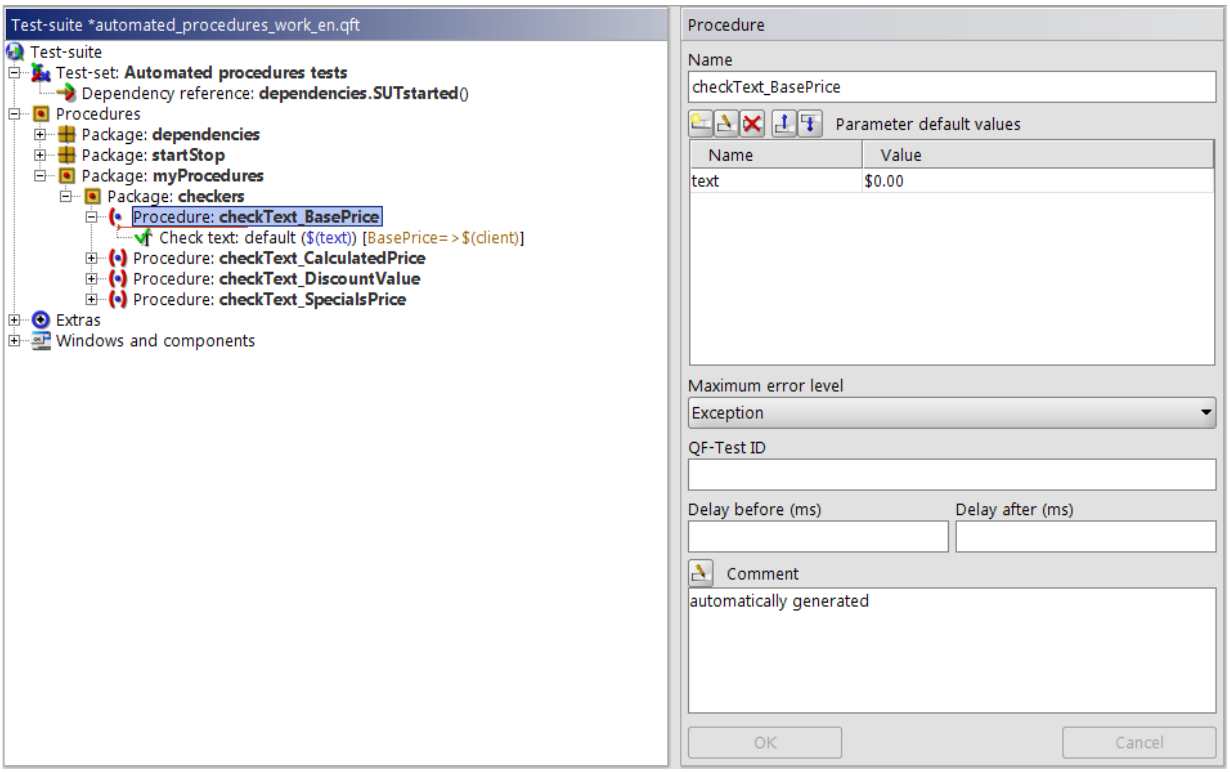

Figure 17.13: The created procedures with the current text

17.3.4 Creating container procedures

In the previous example we have created the 'checkText' procedure for checking single text-fields. Now you should be able to create a 'set' procedure for setting the text of a text-field or a 'press' procedure for pressing a button. Those procedures are created similarly. Keep in mind that all those procedures work with one single component.

In some test scenarios it might be interesting to call one procedure for checking all components of a specific dialog or specific panel or to fill all visible text-fields. Because such procedures deal with components which act as containers, we call those procedures 'container'-procedures.

In our case we could consider one procedure calling all single 'checkText' procedures of the JCarConfigurator. But how to create such a procedure like that?

First of all add a new 'class'-package to your 'mySettings.qft' file with the name 'javax_swing_JFrame'. The main window of the JCarConfigurator is an instance of `javax.swing.JFrame`, that is why we have to use that specific package name. The 'class'-package has to contain a 'type'-package which we call 'checkers-window' now. The 'type'-package has to contain a procedure `checkTextOfElements_<COMPID>`,

intended to perform the actual checks. We use the <COMPID> placeholder again for the procedure name to determine which dialog is affected by the created procedures.

The next step is to specify the procedure content. Let us see how we can configure this.

We have four text-fields which can be checked by using the same procedure checkText_<COMPID>. QF-Test allows to call all four procedures with just one configured Procedure call node. Therefore insert a Procedure call for the procedure javax_swing_JTextField.checkers.checkText_<CCOMPID>.

The configuration file should look like below:

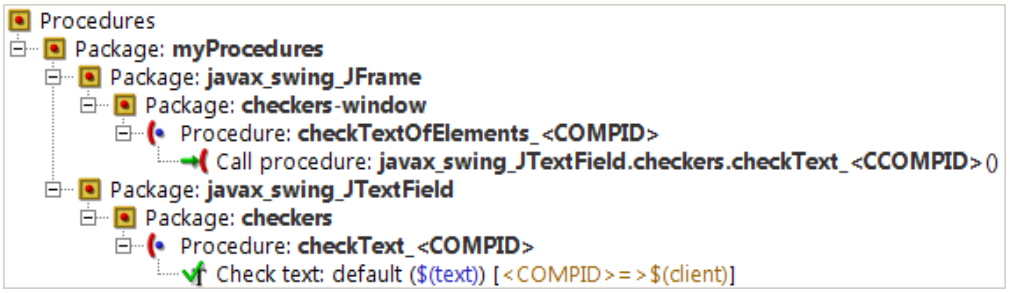

Figure 17.14: The template for container procedures

The last step is to make QF-Test to create a 'container' procedure and not a normal 'component' procedure. This can be achieved by setting @FORCHILDREN in the comment attribute of the checkTextOfElements_<COMPID> procedure.

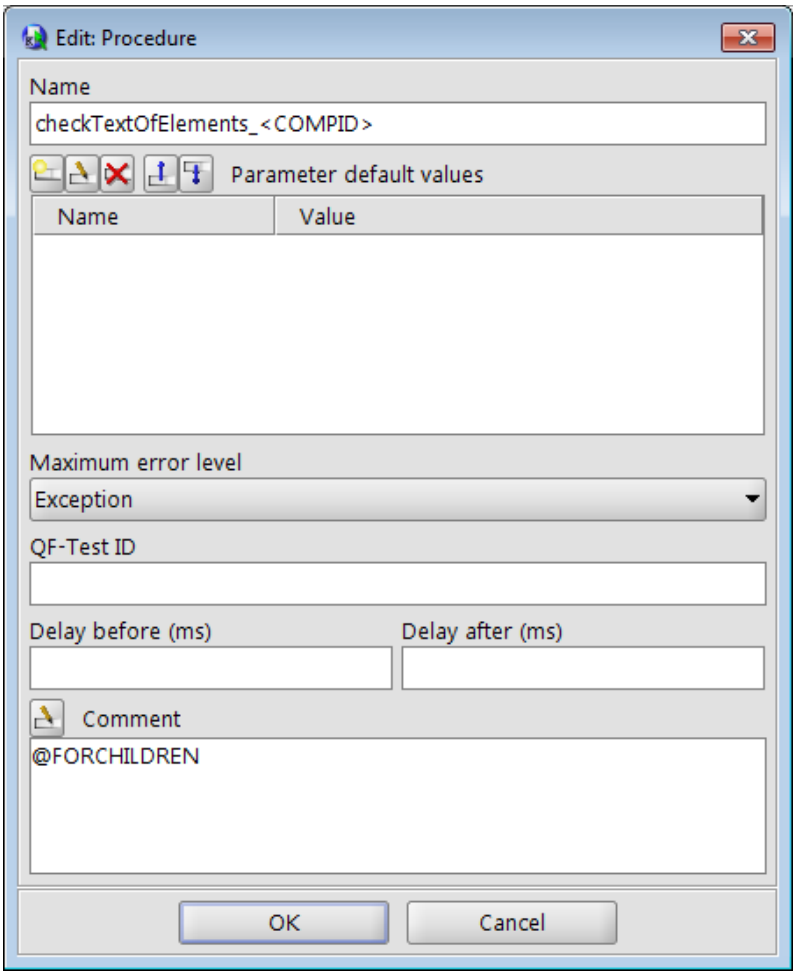

Figure 17.15: Usage of `@FORCHILDREN` tag

Now you can record the procedures like in the previous examples. Do not forget to remove the 'myProcedures' from Procedures before. You should then get something like this under Procedures:

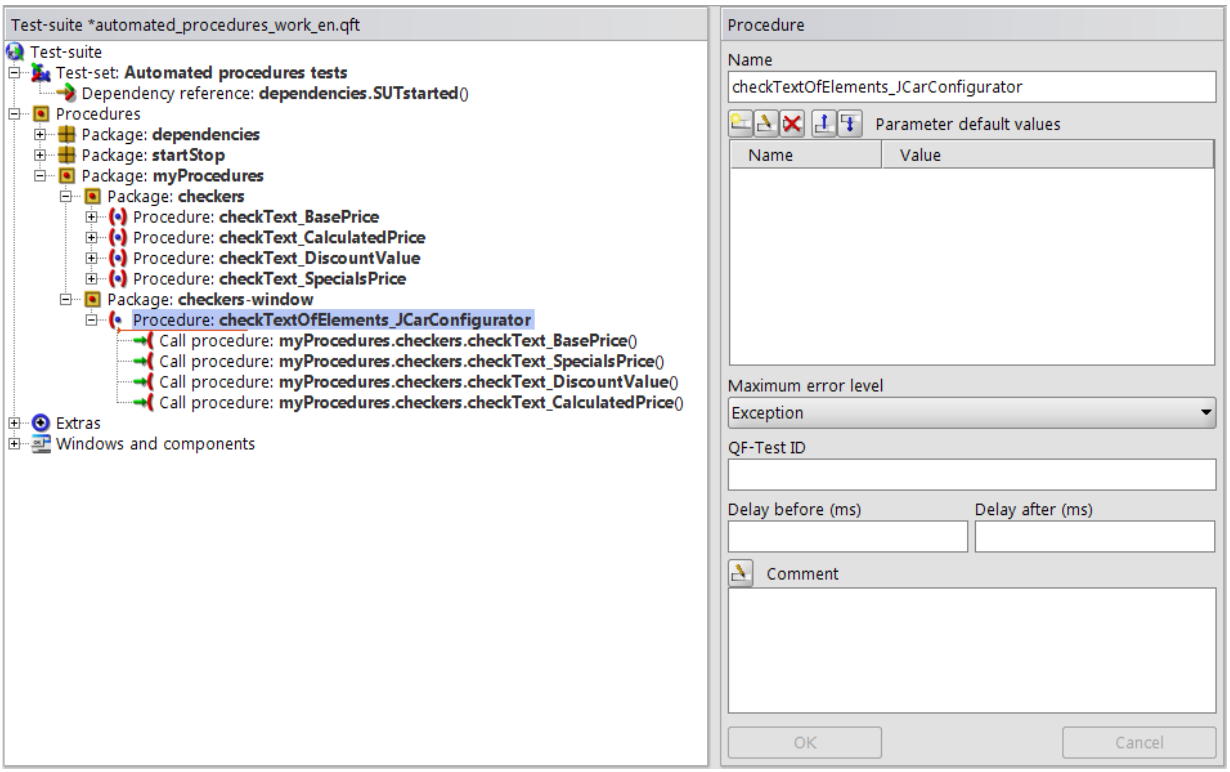

Figure 17.16: The created container procedures

Note QF-Test replaces the 'class' part of the procedure-calls by the top package of the configuration. In our case this is 'myProcedures'.

17.3.5 Using the current value of child components

We could meet a similar situation like in the example Using the current text for checking[201], where we want to record the current values of each text-field during creation. In the example Using the current text for checking[201] we used `<CURRENTVALUE>`. Now we have to specify the current text as parameter at the procedure-call in the 'container'-procedure 'checkTextElements'. Therefore we add the parameter 'text' to the procedure-call in your 'mySettings.qft' file and the value should be `<CCURRENTVALUE>`.

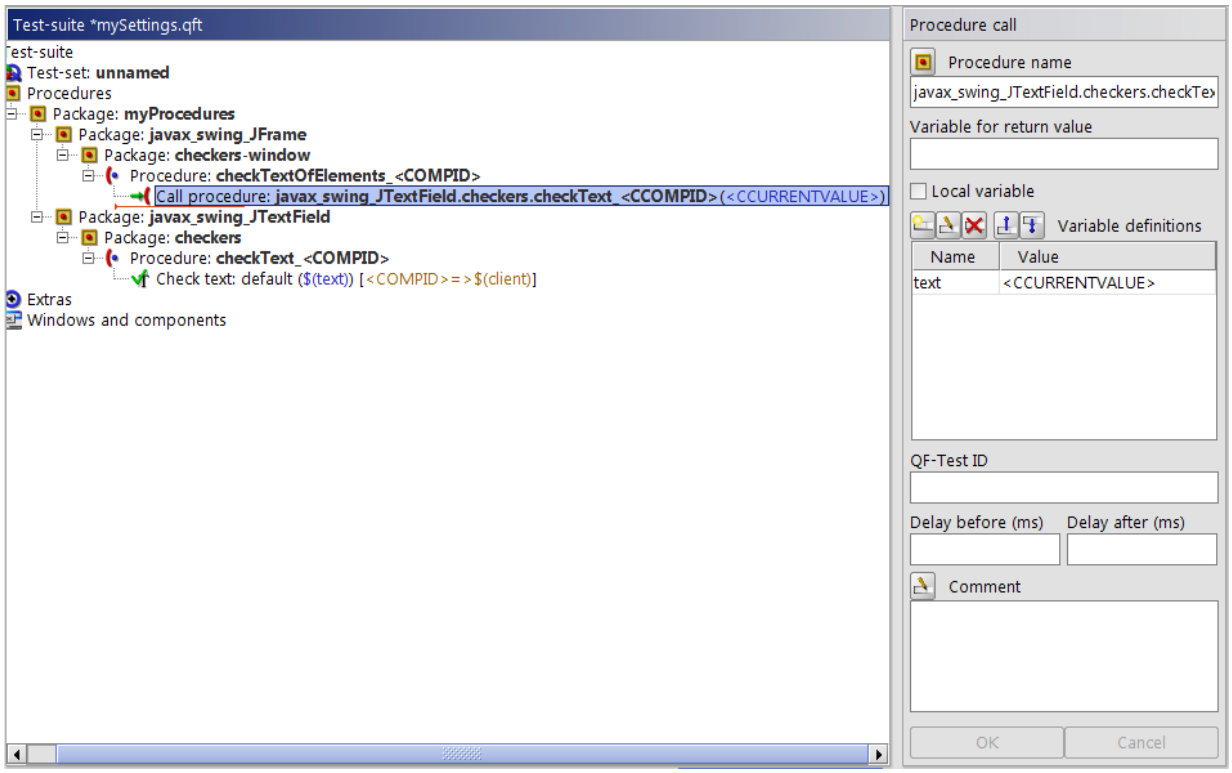

Figure 17.17: Configuration of <CCURRENTVALUE>

If you create the procedures again, you will see the current values at the procedure-calls. Please do not forget to remove the 'myProcedures' package from Procedures.

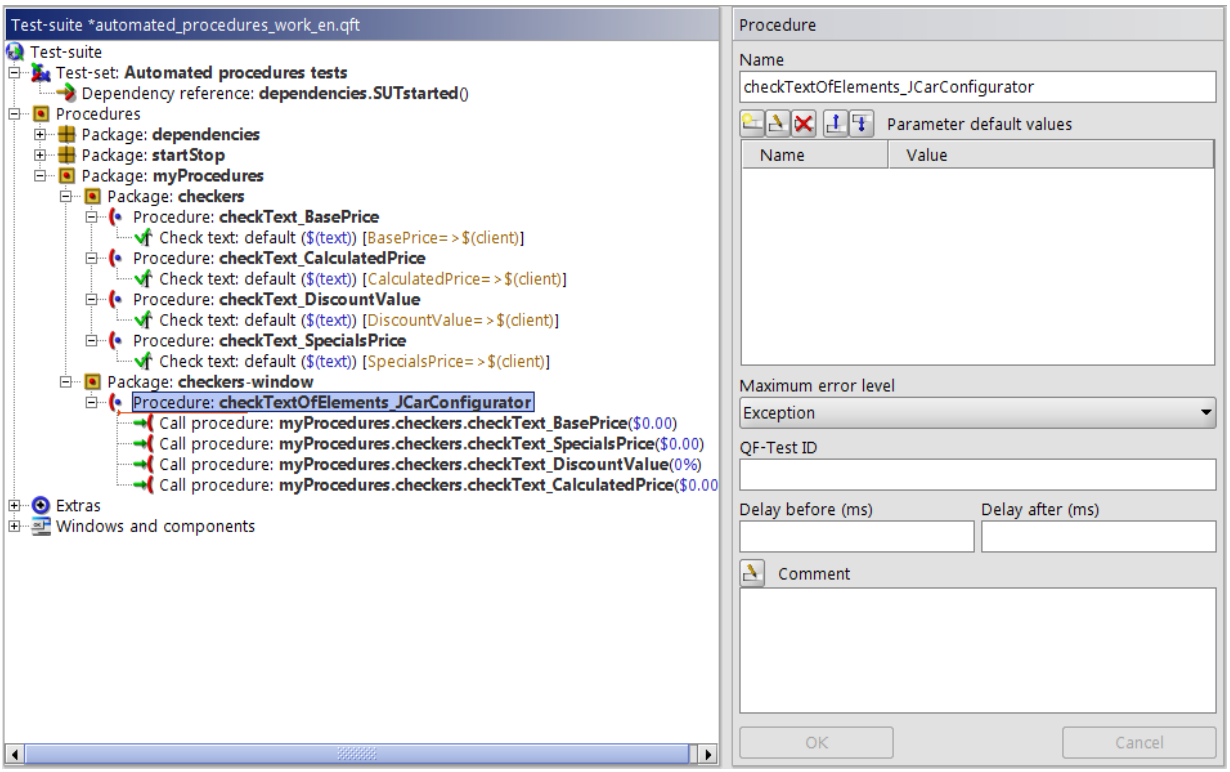

Figure 17.18: Test-suite using <CCURRENTVALUE>

If you take a closer look at the created procedure 'checkTextElements_', you will see that each procedure-call gets the parameter text. Perhaps it might be convenient for test development to set the according 'text' parameters as default values for the 'container'-procedure. To achieve this you have to add one more parameter to the procedure call in the settings file. The name of the parameter has to be <CCOMPID> and the value will be <CCURRENTVALUE>. Then you have to change the value of the text parameter of that procedure-call to $(<CCOMPID>). The configuration should look like this now:

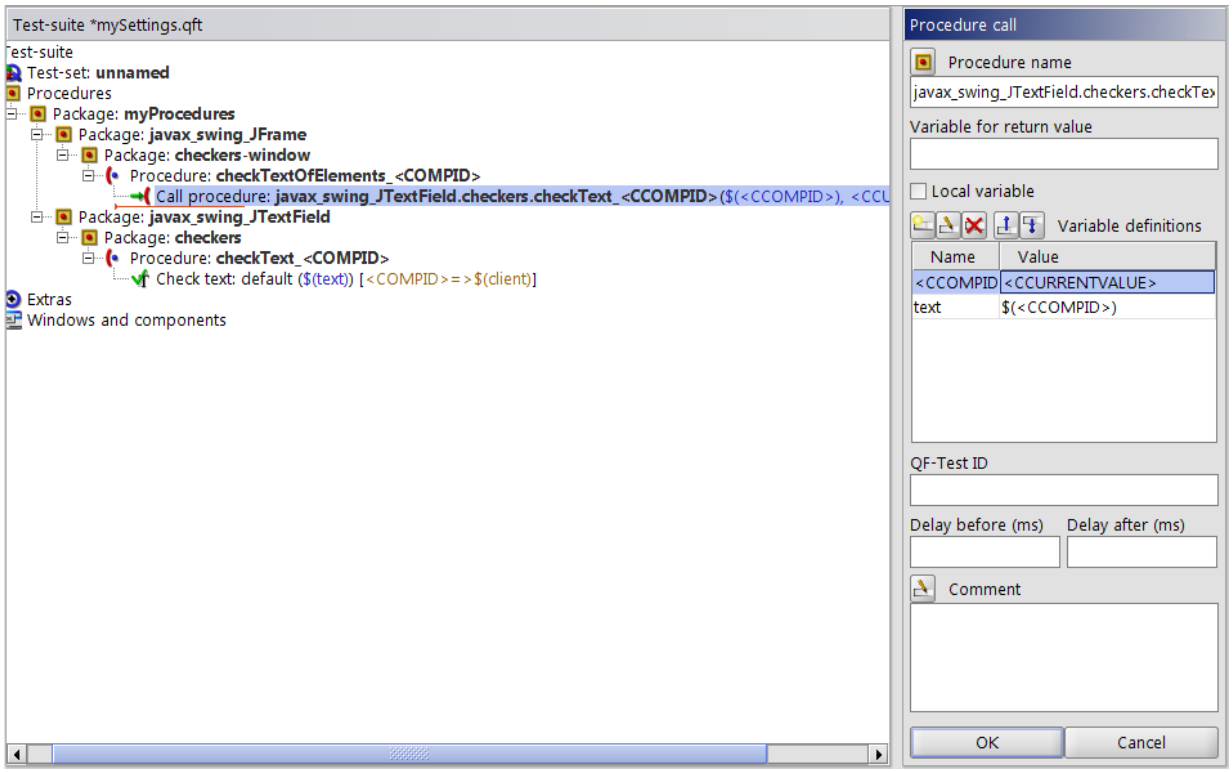

Figure 17.19: Parameters for container procedures

If you create the procedures again, you will see that the 'checkTextElements' procedure has four more parameters with the current value of the elements as default values. Additionally each single procedure call uses a variable - named after the component-id - as 'text' parameter.

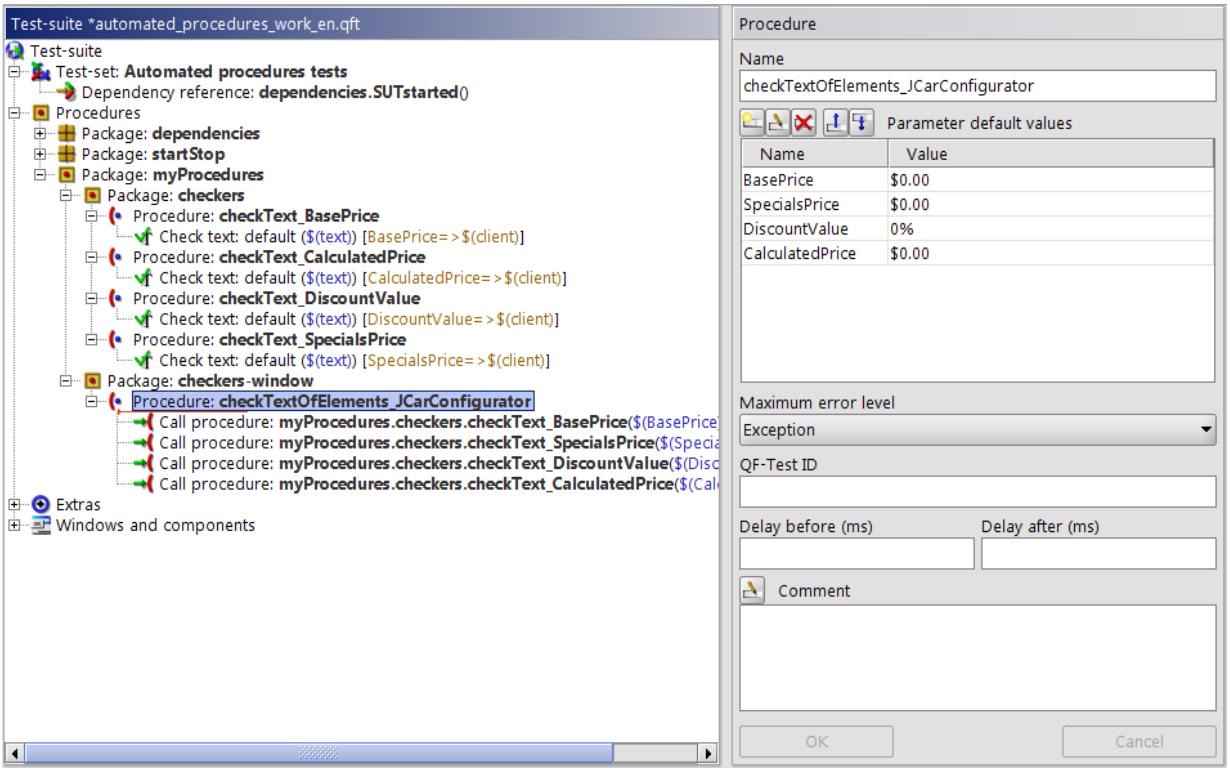

Figure 17.20: Parameters for container procedures in test-suite

17.3.6 More configuration capabilities

As you have seen in the previous sections there is a lot of configuration capability for the automated creation of basic procedures. But there are even more options available. Please take a look at the manual chapter The Procbuilder definition file.